UCLouvain

epl

École polytechnique de Louvain

Building a secure and auditable Personal Cloud

Author: **Laurent Desausoi**
Supervisor: **Etienne Riviere**
Readers: **Axel Legay, Igor Zavalyshyn, Michal Krol**
Academic year 2019–2020
Master [120] in Computer Science and Engineering

Contents

Abstract

Nowadays, personal clouds are becoming more and more popular mainly thanks to the benefits it brings (availability of resources, ease of use, ease of shareability). However, a few decades ago, we never asked a very important question: what are the disadvantages of personal clouds? A few years ago, we started to realise those downsides which were most of the times related to our personal information security. Besides from that, we realised that personal cloud laked the traceability of our information, which means knowing who accessed which data at what time.

The above two concepts became more and more important in modern society, partially thanks to the GDPR. There is a will of the end-users to be able to control their own data. They don't want anyone else to use it for other purposes than legal ones. End-user wants their personal information to be securely stored and that none can breach their privacy.

To cope with all these issues, we propose an innovative personal cloud client called LAUXUS. In this paper, we developed and proved that our client software conceals with the above-mentioned constraints. In a nutshell, we designed and developed a software that allows end-user to harness the power of personal clouds while discarding all the security issues associated with clouds.

Acknowledgements

The first person that I would like to thank is my supervisor for its investment, throughout the entire year (even during the hard time that was the containment). I thank him a lot for taking the time to read again and again my work. Guide me with constructive and relevant feedback during the whole process to have the best outcome possible.

The second person that I thank is Igor Zavalyshyn. He very happily helped me get started with the Intel SGX development by guiding me into which resources were the most relevant and useful. He took some of his private time to help me troubleshoot issues I had for installing SGX and diagnose issues. Always responding kindly and very fast.

Last, I thank my family for its help and support. Especially my father for taking the time to discuss with me about my work (no matter the hour of the day and the day of the year) and by reading, again and again, my work.

These three people are those that made this all works come to a happy ending.

Chapter 1

Introduction

Cloud Storage attracts a lot of interests but unfortunately, the cloud's benefits come at a cost of some security issues (e.g: data theft, data leaks, password leaks [30]). On top of security, users privacy is a subject that is becoming a priority to most people. Among Cloud Technologies, we can distinguish Cloud Storage (e.g: Amazon S3) used by application developers and Personal Clouds (Dropbox, One Drive, Google Drive, etc) oriented for the end-user only. It has been proven throughout the years that Personal Clouds companies like the cited ones suffered some security issues and if not, have been highly controversial considering users privacy.

With this in mind, it could be interesting to outsource the data to the client without outsourcing the Cloud control as per [5]. What this means is that instead of trusting the Personal Cloud to protect our information, we consider it is up to the client to design a protocol that secures his information before giving them to the Personal Cloud. It is similar to the edge-centric concept aborded in [11]. In this way, the users keep only the advantages of the Personal Cloud (mostly the sharing capabilities and the remote storage) while discarding the security issues.

As said in the beginning, users privacy is important in the world of Personal Cloud. Sadly, papers have proven that securely store personal data under the new 2018 GDPR is extremely difficult and requires grounds up solutions (Cfr. [28] and [27]). Furthermore, this also heavily impact real-time compliance. Users needs from storage systems dealing with sensitive personal data to enforce compliance. One of the ideas behind GDPR is that personal data should be seen/used only when it is for a legitimate purpose. GDPR states that users should know how and when their data are being processed. This is something that all Personal Cloud provider lacks, once we share our data with other users we don't know how these users are using them (e.g: once we share a folder with a friend, we might want to know which file in the folder our friend is accessing and for what purpose). This introduces the concept of auditability that GDPR requires: allowing to trace who

accesses which data and for what purpose.

Furthermore, with popular Personal Cloud provider, we are lacking the opportunity to set a rigorous control policy allowing a user to only view or edit a file. This is quite powerful as we often face a real-life situation where we want people to consult our work without being able to edit it.

Following our security scenario (security outsourced at the client side) sharing capabilities becomes an interesting and non-trivial challenge. Indeed, if we secure our information through encryption, how are we supposed to securely share this information with other users? The problem can be split into two:

1. Sharing the encryption key between users. This problem is trivial with an online server through a PKI (Public Key Infrastructure) and the HTTPS protocol. This problem becomes way more difficult when the exchange is between two clients without the aid of external third-party servers.

2. Revocation is extremely problematic with this approach. Indeed, as the encryption key has been previously shared with allowed users, the owner has no other choice but to change the encryption key (as all the allowed users know the encryption key). This also means that the owner must furthermore re-encrypt all the shared information.

An ideal solution for this problem would be that the client never knows the encryption key (the client software never reveals the key to the client). Sadly, no widespread technologies currently exist to enable that.

To answer the challenge of edge-controlled Personal Clouds in the age of the GDPR, we propose LAUXUS: an auditable and secure Personal Storage. This software is run on the client's computer and acts as a transparent layer encrypting information on the fly before uploading them on the Personal Cloud. As stated above, GDPR compliance and edge control are difficult to achieve in systems where we can have malevolent users or malware installed onto their devices (from this perspective, the cloud is more secure than the end-user device). To cope with this issue, we chose to leverage secure computation on the end-user client thanks to Intel SGX Enclaves. Furthermore, LAUXUS has the characteristic to only trust himself (and instances of himself) and shares the storage content only with other instance of itself. On top of that, information handled by LAUXUS never leaves the program itself. The above two functionalities are also a motivation to leverage SGX Enclave.

This paper contributes by developing and designing a protocol respecting the above restrictions. Plus, it also acts as an interesting use case of SGX Enclave in the hope of making Enclave more widespread.

At the time of choosing this subject, no similar work had already been done. In the meantime, a paper wrote by J. Djoko called NeXuS[7] was published. Although

we asked to cooperate on their code (on which they worked nearly 3 years), they preferred to continue working on their own and making the code public one year later (way too late for us to work on it). Therefore, we based our model and protocol on his and re-implement what he had done, which is in itself a challenging work. Furthermore, we went far beyond the objectives of NeXuS with the innovative auditability aspect (which was never discussed inside NeXuS nor in the modern state of the art). Plus, this work also serves as a vulgarisation paper to the NeXuS paper which is very short and extremely technical (from a non-expert point of view), in the hope of helping the growth of SGX Enclaves outside the specialist area. In a nutshell, this work brings life to the first Personal Cloud supporting edge-control (as NeXuS did) but it also enforces auditability and purpose-aware access.

Chapter 2

Problem Description

2.1 Overwiew

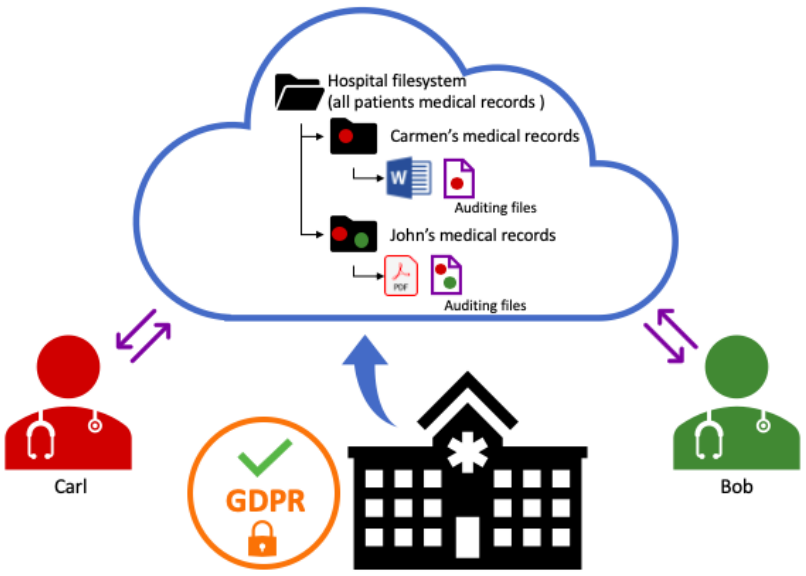


Figure 2.1: Representation of the real life situation

To motivate our work, we are using an example linked to the healthcare domain, more precisely the hospitals. Inside a hospital, many services or doctors must share medical records of a single patient, as depicted in Figure 2.1. How can those parties share this record without disclosing its content (to other parties) and without requiring complicated operation? On top of that, the revocation of a doctor right to access a specific resource, once he no longer needs it, is also a mandatory operation according to GDPR.

Alongside data confidentiality, keeping track of who accessed or edited a given file is required. GDPR states that: "*All EU institutions have the legal obligation to keep a central register of records of activities processing personal data*". In practice, this means that every time an authorised doctor wants to access a medical record, s/he must first explain her/his intention. These intents explain the purpose of the action. These purposes allow a data owner (e.g: a patient) to know who accessed his information and avoid allowed users from abusing his private data. As the purpose may itself contain confidential information, it must also be securely stored. Most of all, these intents can not be tampered with, to keep their accuracy. Furthermore, for a forensic purpose, it would also be interesting to spot if unauthorised users are trying to alter these data. To top it all, these intents must be easily accessible for GDPR compliance evidence.

Aside from the two above functionalities, it is important to note that the users may use portable devices that works entirely offline whereas the concerned user must have a copy of the required documents (e.g: in the event of an out-of-the-office consultation in a remote area whereas there is very poor connectivity or even no internet at all). Furthermore, users must still be able to use their conventional legacy applications (e.g: such as PDF Reader, Microsoft Word, etc) as forcing them to use a new application is difficult and incurs a lot of extra work to the users. We wish to bring no extra complication to the end-users, they should barely see the difference compared to a standard filesystem.

2.2 Use Case Description

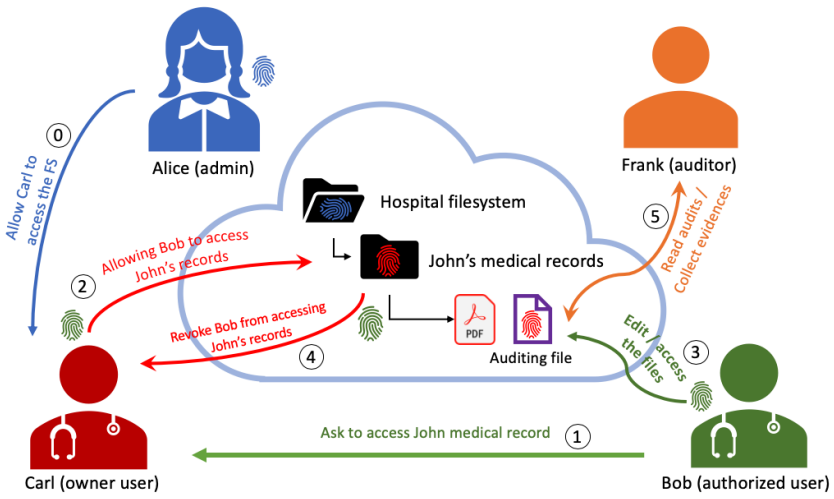


Figure 2.2: Use case representation

Before going deeper into the protocol description we will first take a look at a higher level use case presented in Figure 2.2. First of all, we see that the resource shared between parties is what we call a filesystem. Furthermore, we see that three parties are intervening, each of them having a specific role. To explain each role, we are going to look at the scenario and explain one step at a time what is its purpose and who are the actors.

At first, we have Bob, the authorised user who needs to access a specific file or folder of the filesystem, John's medical record in this case. To do so, Bob must first be authorised by the owner user, Carl (e.g: Bob is a specialised doctor with an appointment with John and Carl is John's general practitioner). Carl will allow Bob to access this directory by linking Bob's fingerprint (the green fingerprint in Figure 2.2) to the target. In this way, Bob can now access the target when representing its fingerprint. Note that this process must be done online, every following action can then be freely done offline (at the condition of having a downloaded version of the Filesystem).

The very same process happens between Alice and Carl beforehand. The role of the administrator (Alice) is to create the filesystem and to upload it to the cloud storage. She is also responsible for the user management: registering and assigning specific roles to each user. Note that, the administrator only deals with which user can access the Filesystem while the owner user decides which authorised user access which file or folder inside the Filesystem.

Once allowed, Bob can do whatever required actions he wishes (in the range set by Carl) at the condition that he specifies the purposes for each of them (e.g: this prevents Bob from accessing John's record outside an appointment and avoid Bob from abusing John's private information). Those intent are stored in an auditing file linked to each accessed file within the target directory. When Bob no longer needs to access John's records, Carl can simply remove its fingerprint from the target folder thereby preventing Bob to access newer versions of this directory.

Lastly, in the case of a GDPR inspection, an auditor, Frank, can retrieve and analyse the auditing files linked to John's records. Theses auditing files list all the operations made by users to a file, along with contextual information like the justification, the date/time and so on.

2.3 Design Goals

After reading the previous sections, we can now establish the different goals that we wish to reach when building our solution.

Practicality: First of all, we want our solution to be transparent to the users. Their typical business workflow should not be affected by the solution. In the same manner, the solution should only slightly impact performance. Data Management

should be fast and easy, at the reach of anyone. This data management refers to allowing a user to access a resource and revoking a user from this resource. Here, we are not considering the complexity of the installation procedure as it is not relevant.

Strictly-enforced access policy: The solution should also maintain a strict and secure access policy. Only allowed users can access the filesystem and an entitlement policy must be put in place to authorise access to each directory and file. By entitlement, we mean the possibility to either read, write or execute the resource (or any combination of these). The importance of these access policies forces them to be encrypted and tamper-evident. This access policy is part of what we aforementioned called "Data Management".

Portability: The usability of this filesystem should not be limited to a single computer. A user must be able to access a filesystem and use it independently of the computer he is using, as long as it has been correctly setup. This means that a user must have some sort of credentials to prove its identity to the system. The end-user can choose to store these credentials in many different ways that we won't discuss here. However, the form and the authentication procedure will be a strong focus in the next chapter.

Confidentiality: Any unauthorised entities will not be able to gain accurate information about the users or the files contained inside the filesystem. For example, the patient's name can be visible inside directories or files name which is the pieces of information we don't want to disclose (e.g: naming a file "*john-tumor.pdf*" divulge personal information that could be used by malicious users). However, we are not concerned about disclosing the structure of the filesystem hierarchy (e.g: knowing that under the root directory there are 10 files and 3 folders). This confidentiality goal implies that our cryptographic solution (the algorithm used) must be strong.

Purpose-aware access and write-only auditing files: Every action took by an authorised user, may it be reading/editing, must be recorded along with a justification, before being able to proceed with the action. In practice, this means that when Bob wants to save his Microsoft Word document to disk, a prompt message should appear asking him why these modifications were needed (e.g: update of the medical record, new insights about a patient, etc). These justifications must be secured and available to adequate parties immediately. Needless to say that these auditing files must be tamper-evident and no one, except the adequate parties, can read them. Even adequate parties can't change them.

Security: The solution shouldn't trust the user, together with its device (at the exception of the SGX Enclave in the device's CPU). The reason for not trusting the users is to protect the Filesystem against users who are wishing to read or edit files they are not supposed to access. Not trusting the user's device means that even if an attacker has access to the computer (e.g: through some kind of

malware of physical access), he can't retrieve any sensitive information about the Filesystem.

2.4 Why is it a challenge ?

In this section, we are going to analyse the problem according to what is currently available in the state of the art. To do so, we will look at a naive solution and demonstrate its limits (and thus prove the limits of current technologies).

The solution could be to simply encrypt, with a secure key, the filesystem and then upload it to the remote storage. When the owner wants to share this with someone else, he can simply give him the encryption key and the intended user can securely decrypt the content. This can easily be scaled up to any number of users to share with. This solution has limitations:

Access policy becomes impossible using this approach. Once the owner transmits the encryption key to a user, this user can decrypt the entire filesystem. No granularity can be made by the owner. A possible workaround for creating this granularity is to encrypt each file with a unique different key. Thus transmitting then multiple keys when the filesystem is shared.

Beyond the scalability issues, this leads us to the **user revocation** problem. Indeed, the only way for the owner to revoke users access is to generate a new key for each file the user had access to. The owner must then re-encrypt the entirety of the concerned files and transmits all the newly generated keys to the remaining authorised users. This means that the revocation procedure has a huge overhead, scaling up proportionally with the number of files the revoked user has access along with these files size.

Most of all, **encryption keys protection** is impossible in this situation. All the authorised users know each of the keys they have been given and it is up to them to chose whether or not they want to share this key with others.

Lastly **transparency** can only be achieved by using a layer intercepting IO system calls (which means that the user's device must be trusted to deliver the correct data to our software). In this way, the approach works no matter the user-space application the end-user is using. This layer allows us to manipulate the data saved to disk however we want. Also, this layer is the core mechanism of our solution allowing to make all required operations for our filesystem to be at the same time confidential and auditable.

2.5 Threat model

We will now look at the threat model of our system. To do so, we will use the STRIDE model developed by Microsoft[32] instead of using an *ad hoc* approach. STRIDE model allows us to be more structured and cover every security aspect that could be forgotten using an *ad hoc* approach. We will define each aspect of STRIDE[1] for each party acting in our system, which are: the **Cloud Provider**, the **auditor**, the **administrator**, the **data owner** and the **authorised user**.

First, we are going to look at the Cloud Provider. We consider him as an untrusted party to which no information can be disclosed and that is not trusted to hold any confidential data. We only consider the Cloud Provider to be honest-but-curious, in the sense that he must correctly share data to users. With this in mind, the **Denial of service** aspect becomes irrelevant. As well as the Cloud **Spoofing** because an attacker must still correctly implement the sharing protocol.

Second, we consider all the other entities:

Spoofing any user is a mechanism whereby a malicious actor impersonates our authorised user. This must be prevented as much as possible but as we will see later on, it is only feasible to a certain extent.

Tampering is relevant to all the parties. We must be able to detect when anyone is trying to change the content of a file to which he has not access[2].

Repudiation is only relevant for the authorised user. This must prevent the user from denying their action on the files. Related to this, we must enable forensic analysis indicating that a user tried to do something that he is not allowed to (e.g: tamper with a file).

Information disclosure is crucial in our scenario, however, becomes irrelevant with an authorised user. There is no way to build a system that prevents authorised users from making the information public.

Denial of service means that the user voluntary stops to download or upload information to the cloud storage. This is considered irrelevant here because we consider every authorised user to be honest. As with the Cloud Provider, this means that they follow the protocol, which means that they upload or download information as soon as they can.

Elevation of privilege for all the users means that they can't change their role. Concerning the authorised user, this also means that he can't, in any way, change any entitlements.

[1] Spoofing - Tampering - Repudiation - Information disclosure - Denial of service - Elevation of privilege

[2] This also refers to users that are authorised to access a certain section of the filesystem and that wishes to tamper a section they don't have access to.

A synthesis of this threat model along with a reminder of the role of each party can be found in the following Figure. Green ticks indicate that the corresponding risk is present for the corresponding actor and that we must find protections against this risk. In the other hand, red crosses indicate that the corresponding risk is not relevant for our solution and thus will not be discussed.

	Cloud Provider	Auditor	Administrator	Owner user	Authorised user
Role	Share the Filesystem between the users	Verifies the reason for every user's action	Defines which user can access the Filesystem (the user management)	Defines which user can access which file (the entitlement)	Access allowed files and directories
Spoofing	✗	✓	✓	✓	✓
Tampering	✓	✓	✓	✓	✓
Repudiation	✗	✗	✗	✗	✓
Information disclosure	✗	✓	✓	✓	✓
Denial of Service	✗	✗	✗	✗	✗
Elevation of Privilege	✗	✓	✓	✓	✓

Figure 2.3: Threat model

Chapter 3

Preliminaries

As stated in the previous chapter, to solve our problem we must look at newer technologies that are not currently widespread. The two technologies we need are: one for creating some sort of secure environment inside our computer (to securely manipulate cryptographic material), and the other one is a layer being able to transparently intercept filesystem operation to change their behaviour.

The above-discussed technologies are respectively TEE (Trusted Execution Environment) and FUSE (Filesystem in Userspace).

3.1 Trusted Execution Environment

TEE is essential in our work as it allows for secure computation inside the end-user computer. This means that our software can freely manipulate cryptographic keys without disclosing them to the user or the potential malware on the machine. In this section, we will look in more details what is a TEE and how it works, especially the SGX Enclaves that we are using in our work.

3.1.1 Definition

A Trusted Execution Environment (TEE) is a secure area of the main processor which offers an isolated environment running in parallel to the main operating system. This isolation allows secure computation which means that: the code, the static data and the run-time states (CPU registers, memory, etc) are kept confidential. On top of confidentiality, TEE guarantees code and static data authenticity. It embeds secure storage and remote attestation capabilities to prove its trustworthiness to third parties applications.

3.1.2 Security requirements

Following the above definition, it means that TEE must resist to all software attacks and physical attacks on the main memory coming from outside the TEE. Furthermore, exploiting backdoor security flaws must be impossible.

The foundation of a Trusted Execution Environment is the separation Kernel. The goal of a separation Kernel is to allow the coexistence of systems of different security requirements on the same platform. In a nutshell, a separation Kernel divides the platform into multiple partitions guaranteeing isolation between each one of them. The only exception is the inter-partition interface enabling inter-partition communication. As stated in [25], the separation Kernel has the following security requirements:

- Data separation: data within one partition cannot be read or modified by other partitions.
- Temporal separation: shared resources can't be used to leak information to other partitions.
- Control of information flow: Communication between partitions can only occur if previously permitted.
- Fault isolation: Security breaches in one partition should not affect the others.

TEE security is described by its Trusted Computing Base (TCB). This TCB is, in fact, the size of the code run inside the TEE. Further discussions about TEE and their building blocks are beyond the scope of this work and can be found in [25].

3.1.3 Available technologies

Among TEE technologies, the two most widespread are ARM TrustZone and SGX Enclaves. The following comparison helped us choose which TEE technology choose for our solution. Both are quite different, their main difference comes from the architecture they run on. Concerning TrustZone, it is designed only for ARM architectures which are mainly used for mobile and micro-controller devices. Historically, it is associated with single-purpose systems (only one TrustZone per device). On the contrary, SGX Enclaves were designed for multi-purpose chips where the system's purpose wasn't known at chip design time. This design choice allowed SGX to have the potential to run multiple enclaves (with each a different purpose) on the same system.

The latter technology was chosen to create our solution thanks to its multi-purpose ability enabling an end-user to run other Enclave if he wishes so.

3.1.4 Intel SGX Enclaves

Intel SGX Enclaves are a type of TEE developed by Intel in 2018. It works by providing a set of security-related instructions into modern Intel CPUs[1]. This allows the creation of an encrypted and secure memory region called an Enclave. This region is protected from any other user on the platform. In a nutshell, an Enclave considers that everyone else is a threat except the Enclave itself.

SGX implementations are based on a very simple principle: there are two worlds, a trusted one and an untrusted one. Each of these two worlds possesses its data and code. Every exchange between these two must be precisely described through an interface. Intel uses two concepts to describe these interactions: either an ECALL (a function call into the secure world) or an OCALL (a function call into the insecure world). These two concepts are used to define the interface of the application (its Trusted Computing Base).

The main drawback brought by Intel SGX Enclaves is their limited memory. In practice, all the Enclave running on a computer must share around 100MB of memory. This limitation incurred some limitations in our solution but was reasonable and with more development, it can be easily avoided (Cfr. Section 5.3).

Memory Management

Memory Management inside an Enclave may sound complicated but the main idea is straightforward: every data exiting or entering the secure world of an Enclave must be encrypted as can be seen in Figure 3.1. The information inside an Enclave is stored inside the EPC (Enclave Page Cache) which uses the MME (Memory Encryption Engine) to encrypt the pages. This MME is a new and dedicated chip for this purpose. This means that reading information on the memory bus will only result in observing raw encrypted data. The EPC can only be decrypted inside the processor core thanks to keys generated at the Enclave creation.

Trust establishment

For an Enclave to establish trust, there are three main activities:

- **Enclave Measurement:** Enclaves are accompanied with a self-signed certificate provided from the enclave author allowing the Enclave to detect whether any portion of the Enclave code has been tampered with. Unfortunately, this is not enough as it only authenticates the initial state of the Enclave, not the running state. To cope with this, two more information are provided within the certificate.

[1]SGX Enclaves must be enabled inside the BIOS which is not the case for many server-grade systems.

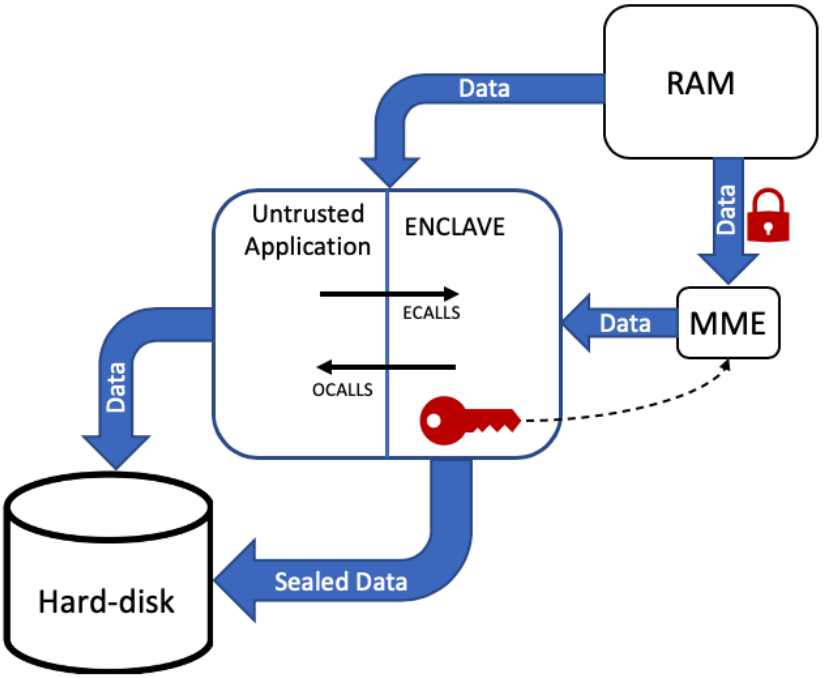


Figure 3.1: SGX Memory Management

First, a measurement 256-bit hash that identifies the code and the initial data loaded inside the Enclave. When the Enclave code and data pages are placed inside the EPC, the CPU calculates this measurement. Second, there is also a hash of the Enclave Author's Public Key. This last information allows enclaves authenticated with the same public key to securely speak to each other.

- **Data sealing:** When an enclave is instantiated, as discussed above, its data and code are protected inside the Enclave. Unfortunately, they can't stay forever, when the Enclave stops, it must be able to encrypt these pieces of information into untrusted storage. This process is called the sealing procedure and in most cases, the untrusted storage is the hard-disk of the computer. SGX Enclaves use a sealing key that can be derived even if the Enclave is stopped at some point in time. There are two manners to derive it: either using the Enclave Identity (the Enclave measurement) or by using the Signer Identity (the hash of the Enclave Author's Public Key). In both cases, two distinct enclaves will derive two different keys. The main difference is that by using the Signer Identity, two versions of an Enclave share the same key and thus can read the sealed data of each other. On the contrary, using Enclave Identity, the above operation is impossible but it may be an

advantage, as it disables the migration of data between multiple Enclaves.

- **Enclave attestation:** An Enclave may need to prove to a third party that it is legit. Intel developed two ways: either local attestation (to another process on the same computer) or remote attestation (to a remote server). We will only focus on the latter attestation procedure as the first one is not relevant in our work. To prove its identity, an Enclave produces a so-called quote which is a credential that reflects the enclave state. This can be done thanks to an architectural Enclave build by Intel called the Quoting Enclave (QE). This quote can then be verified by Intel web-service to check its authenticity. A quote is composed of many information including the Enclave measurement, user-defined data and others. All of this is powered by an anonymous algorithm known as the Intel(R) Enhanced Privacy ID (Intel(R) EPID). This custom scheme allows using a single private key to verify the authenticity of any number of signers in a group (which each uses a different private key). This process implies that the verifier doesn't know which member of the group signed a quote but can verify its correctness. Obviously, in the SGX context, this EPID group is the collection of SGX enabled platforms.

3.2 *FUSE*

FUSE stands for Filesystem in User Space. It is used to override the typical behaviour of the UNIX Filesystem. By its ease of use, *FUSE* allows to easily create personalised filesystem without modifying the OS. It is especially used in the research world to develop and prove innovative filesystem. Some papers have already been published with the idea to protect a filesystem from the cloud such as SGX-FS[4] and SafeFS[23]. But it is not limited to creating a secure filesystem as per [21].

FUSE consist of a Kernel module and a user-space library. It works by forwarding the system calls from the kernel into the user-space library for custom behaviour. A *FUSE* instance is launched by mounting it to a given directory. This directory will then mirror the behaviour described inside the user-space library. A great advantage of using *FUSE* is that it can be used as layers on top of each other (e.g: combine a filesystem that turns the path in lower case and one that creates a directory when the filename is composed of a "-"). Another often favoured advantage is that it is very easy for a lambda developer to create a custom Filesystem thanks to its user-friendly user-space library.

Lastly, we can note that there are two versions of the user-space library: either a low level one or a high level one. The main difference is that the first one works

with inode whereas the latter one works with filenames (and performances are slightly different in the favour of the low-level API).

In a nutshell, the advantage of *FUSE* is that it provides transparency and that a *FUSE* instance doesn't impact the other *FUSE* instances. Its main drawback is its performance: [29] and [24] found out that optimised *FUSE* can perform within 5% of native filesystem. However, every custom file-system might not be friendly with *FUSE* which degrades drastically its performance (up to 83%). Furthermore, it has been proven that *FUSE* file-systems are CPU intensive (an increase of 31%). However, this performance drawback is not important as this work is more focused on security than performance. In our case, performance is acceptable even if it is 3 times slower than standard filesystems.

Chapter 4

LAUXUS

4.1 Approach overwiew

Now that the previous introduction sections, we can start to explain the solution we designed to address the problem. In this section, we will explain the solution at a high level. Figures 4.1 to 4.3 present this protocol and will be the heart of this section. Detailed insights will be discussed in Chapter 4. The approach will be split into four parts to ease the understanding of each major point of the protocol. The whole protocol is also presented at the end of this section.

4.1.1 Trust overview

As presented in the use case Section 2.2 the protocol happens between multiple parties (the administrator, the auditor, the owner user and the authorised user). As a quick reminder, all the relations of trust between each of the roles are previously presented in Figure 4.1.

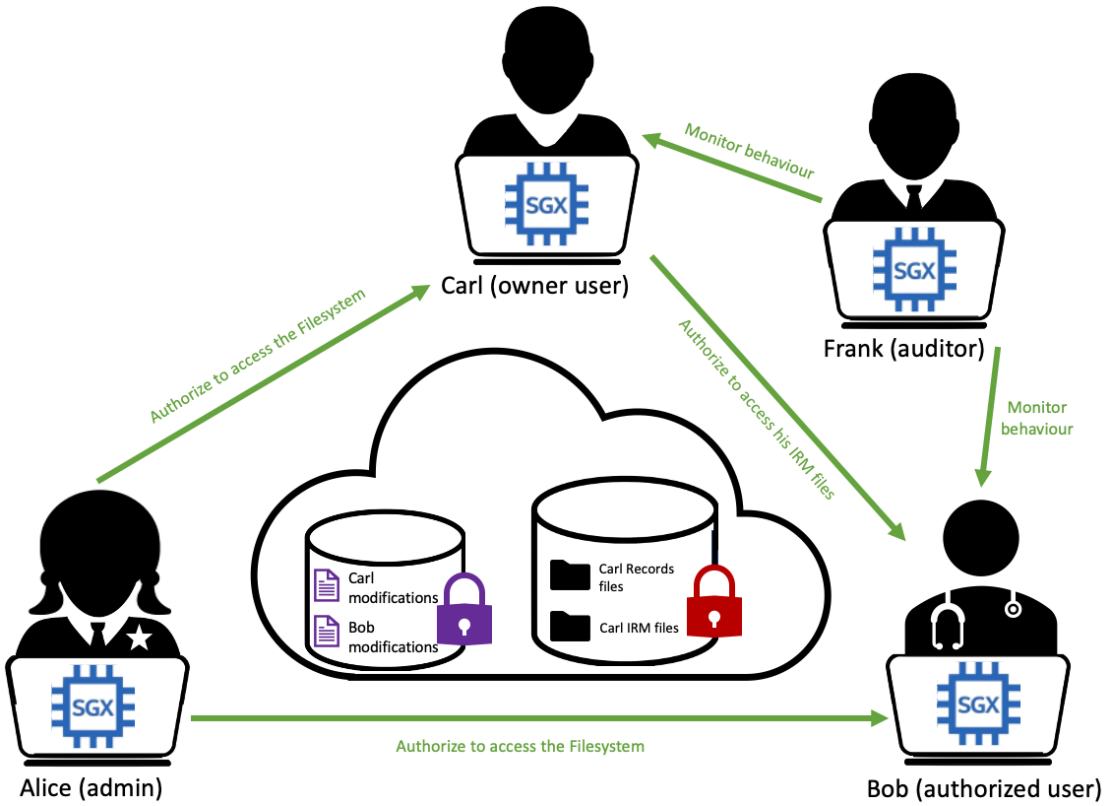


Figure 4.1: Trust overview of the protocol to share the FileSystem

4.1.2 Filesystem overview

Every user is running *FUSE* to interact with the decrypted Filesystem if they are authorised. The *FUSE* interaction is the core of the untrusted application (happening outside the Enclave). Encrypted information received from *FUSE* is then transmitted inside in the Enclave to be decrypted and processed. In the case of reading a file, *FUSE* sends into the Enclave the encrypted content and the Enclave returns to *FUSE* the decrypted content which is then transmitted to the end-user. The same principle applies upon write operation but in the opposite way: sending decrypted information into the Enclave and outputs the corresponding encrypted information to the remote storage.

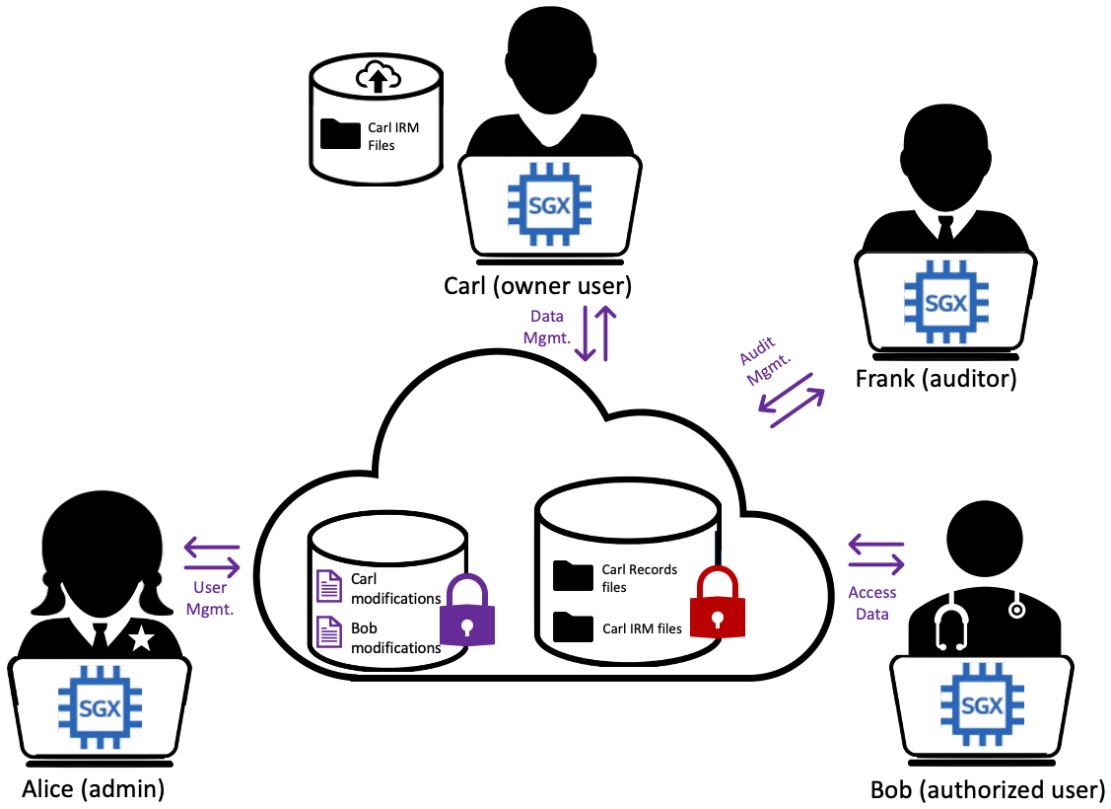


Figure 4.2: Filesystem overview of the protocol to share the FileSystem

4.1.3 Cryptographic overview

We will now take a look at the core of the filesystem, which is its content. To protect confidential information against attackers, the administrator (Alice) will encrypt the filesystem with a secret key (the root key, in red). Then she will upload the encrypted filesystem to the remote storage. To avoid sharing the root key in clear, she will only share it after encrypting it with another key (the shared root key, in blue). This last key will only be transmitted to authorised users through a secure out-of-band channel. This is where we can see the power of SGX and why we chose it: being able to manipulate a secret on a client's computer without revealing any information on this secret to the client itself. Here, the secret we are talking about is the root key. In this way, the intended user, for example, Carl, can decrypt the key located on the storage and retrieve the root key. Then he can access and decrypt the filesystem. After the root key is securely shared between users, it is securely stored inside the user's computer by leveraging SGX Sealing capabilities.

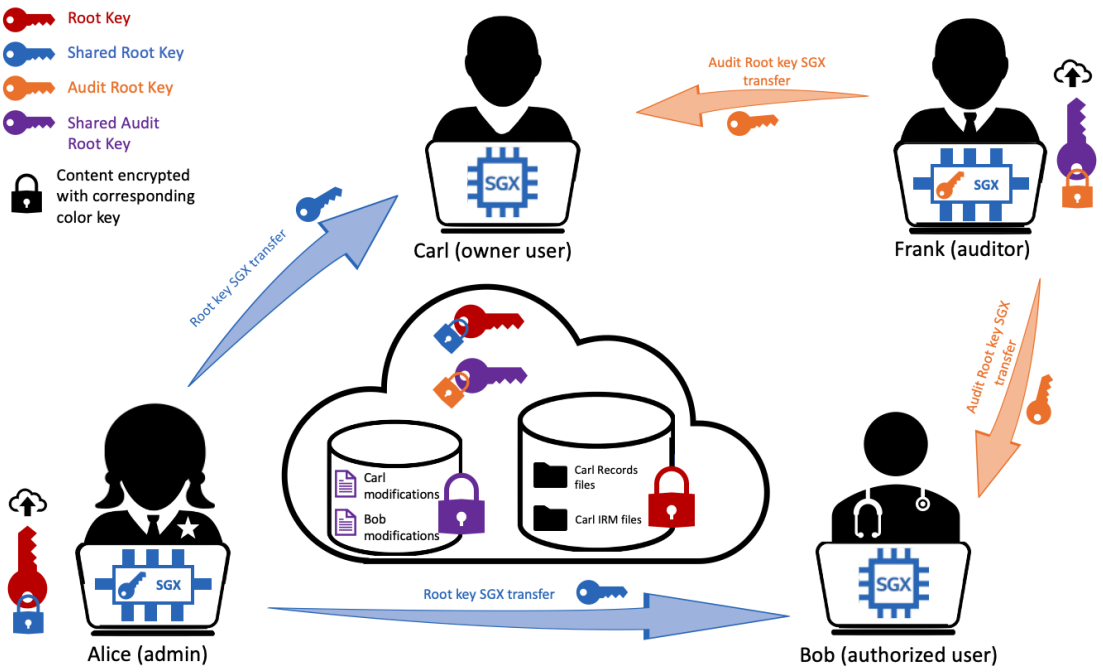


Figure 4.3: Cryptographic overview of the protocol to share the FileSystem

4.1.4 Auditing overview

Finally, we have the auditing files which stores the justifications for every action done by the authorised users along with more contextual information (e.g: the time, the date, etc). There is a 1-to-1 mapping between the auditing files and the files in the filesystem. In practice, the filesystem will create one entry in the appropriate audit file on each read or write of any user, no matter its role. We are using two different keys to have segregation of duties (the administrator manages **only** the filesystem and the auditor manages **only** the audits). The idea is that an organisation, here the hospital, can't at the same time read in plaintext the filesystem content and the auditing files. We don't want an organisation to change the auditing file at their advantage. As audit files are used for GDPR purposes, only a trusted entity should read those, thus owning the audit root key. They work similarly than the filesystem content. Here, the auditor is equivalent to the administrator. Thus, the auditor owns the audit root key (the purple key). The sharing of the audit root key follows the same procedure as with the root key. As explained above, the audit root key must also be securely shared to have a working filesystem.

4.1.5 Global overview

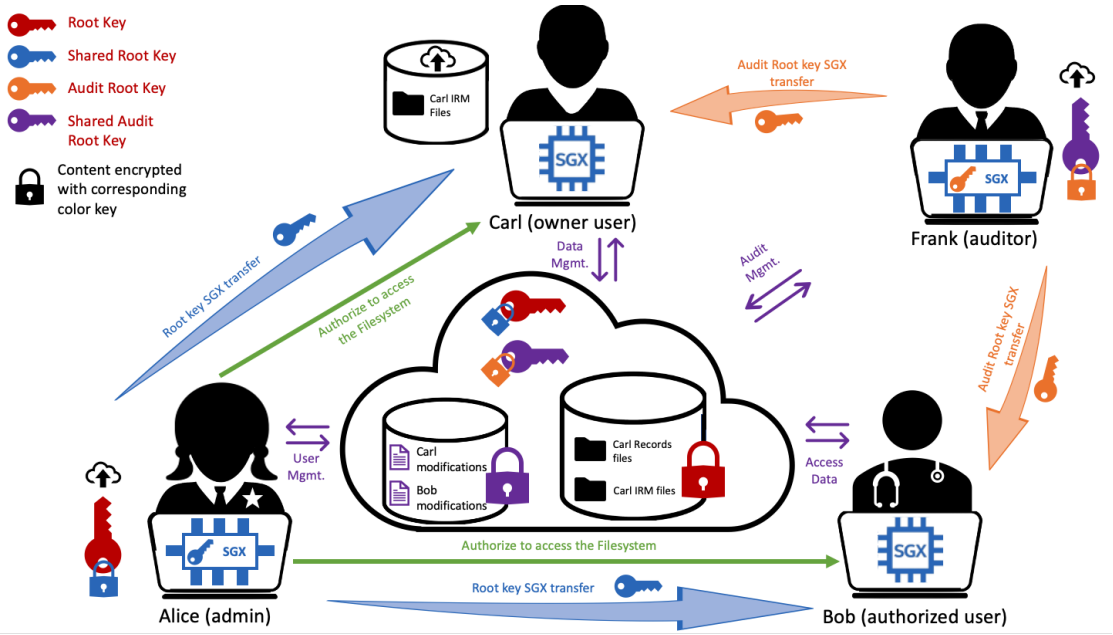


Figure 4.4: Global overview of the protocol to share the FileSystem

When all the above mentioned concepts are merged, we end up with Figure 4.4. As it appears on Figure 4.4, it is crowded and hard to understand at first sight, which is why we choose to split it into four parts.

An UML state diagram can be found in Appendix A for even more detailed information on the whole interaction of the solution.

4.2 Architecture

LAUXUS is a software composed of two planes: a data plane and a management plane. The first one is used to simulate a filesystem to which the user-space application interacts. The latter is used to manage the users accessing the filesystem. An illustration of those two planes can be found in Figure 4.5. In both planes, information is securely stored inside the shared filesystem, in three respective folders: one storing the end-user files, one for the auditing files and a last one for the metadata files. These metadata files enable the filesystem to be shared securely without disclosing information to unauthorised parties.

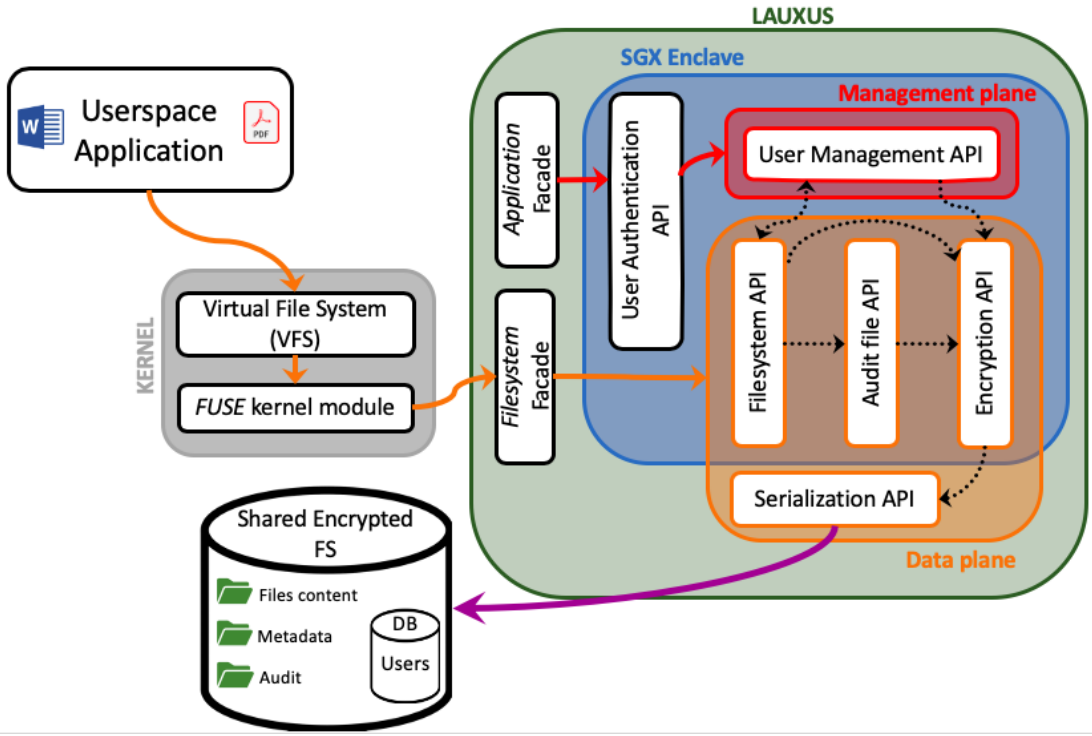


Figure 4.5: Data plane architecture of LAUXUS

4.2.1 Interaction between planes

In Figure 4.6 is represented the high-level flow followed by LAUXUS when launched. As we can see, we first go through an authentication process. Second, depending on our role (Administrator / Data owner / Authorised user), we are redirected to either the Management process (with the management plane) or the Filesystem process (with the data plane). Each of these three processes will be covered in details in later sections. First, we will decompose each plane and explain the functionalities of each API presented in Figure 4.5.

4.2.2 Data plane

This plane leverages the *FUSE* kernel module to intercept all filesystem operations happening at the kernel level. This allows LAUXUS to be fully transparent for the users and highly increases its portability. The user-space application will interact in the same manner with the Virtual File System as if LAUXUS wasn't even running. In order to communicate with *FUSE* kernel module, we must implement a *Filesystem* facade. The *FUSE* kernel module is composed of a little bit more than 20 endpoints (IO system call) which 15 of them have been implemented in

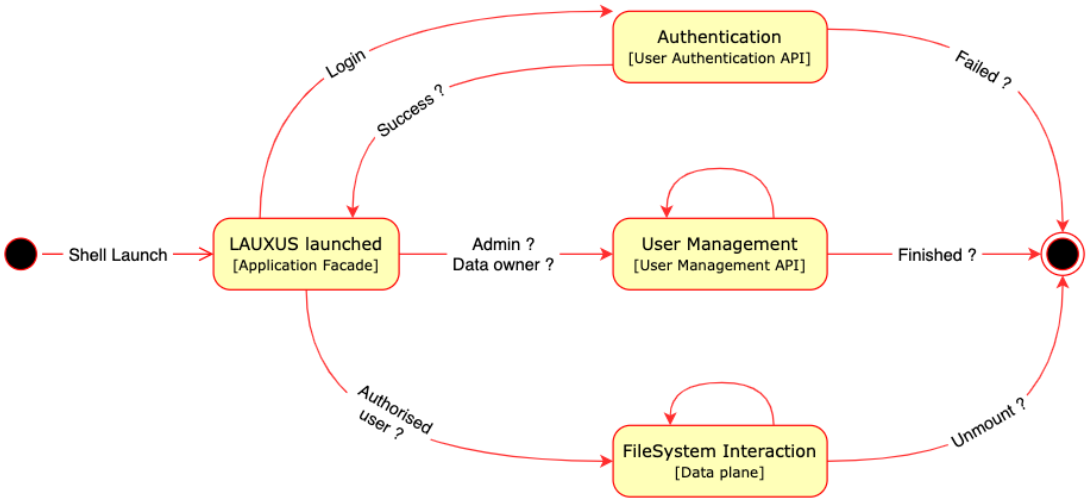


Figure 4.6: Interaction between planes

our facade. This means that some filesystem operations will not work but they are not mandatory in order to have a working Filesystem or for running standard filesystem software (e.g: tar, cat, cp, etc). This facade will then interact with the core of our application inside the SGX Enclave which is composed of multiples APIs:

- Filesystem API: It is used to simulate a real filesystem and responds to each filesystem operation requested by *FUSE*. It also interacts with the Management Plane for the user entitlement (rights to access a specific file).
- Audit File API: It is used to generate and handle audit entries (the purposes of each authorised user action).
- Encryption API: It is used to encrypt or decrypt information respectively before saving them to the storage or before using them inside our others API. This layer allows information to be securely handled.
- Serialisation API: It is used to interact with the storage (either saving or retrieving information). This layer is outside the Enclave as it no longer handles secret information. Indeed, all the data have been encrypted in the layer before.

4.2.3 Management plane

This plane is using an *Application* facade to interact with the core of the application. This facade partially allows the administrator and the owner users to manage the

user database (user registration/revocation and/or user entitlement). This facade is also used for the authentication process but will be discussed in Section 4.6.2. As the data plane, the management plane is composed of multiple APIs:

- User Authentication API: It is used to authenticate users into the filesystem. This login will set up a state inside LAUXUS that will be used by the Filesystem API to check the user entitlement.
- User Management API: It holds the state of the currently connected user and is used to edit the user database. Interaction with the Users Database is secured with the encryption API.

4.3 Filesystem

Now that we have seen the different components of our solution, we can start to look in more details into one of the components of the Data Plane: the Filesystem. The filesystem we designed in this section can have some similar features with the **ext2** Linux Kernel filesystem (inode being similar with nodes described below).

4.3.1 Nodes hierarchy

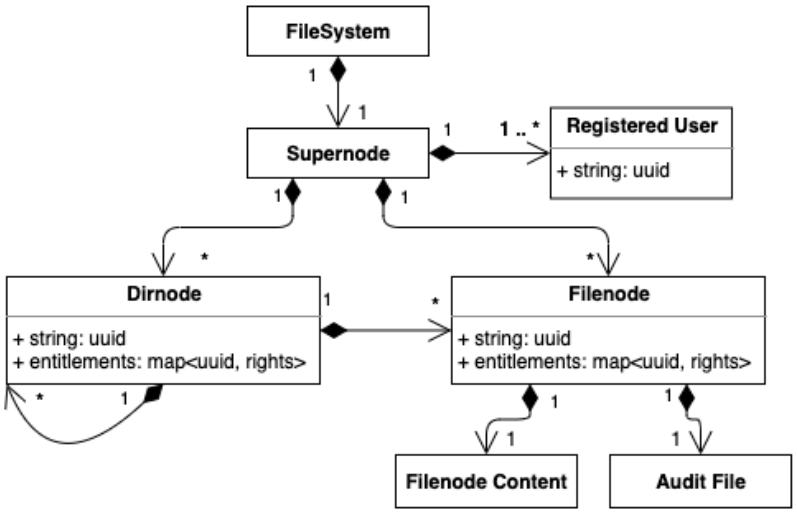


Figure 4.7: Nodes UML hierarchy

Our filesystem is composed of different nodes, each one of them representing a standard filesystem entry: either a directory or a file. Every node is stored and identified by using a random UUID so that, a curious attacker can't deduce any

information about the node's name (e.g: directories named with patient's names in the case of our scenario). Thus, our filesystem hierarchy, presented in the following Figure 4.7, is composed of:

- A single **supernode** acting as a user database. It stores a list of all the registered users. This supernode is mainly used as the first firewall for unauthorised users. A secure handshake is initially made before decrypting and opening the filesystem. This handshake and the user management will be covered in its Section 4.6.2. Along with this, the supernode is acting as the root directory of the filesystem. Thus, it stores a list of its children nodes. Its children may either be a dirnode or a filenode.

- Multiple **dirnodes**, each representing a standard directory. Each dirnode contains a list of user entitlements describing the user rights to access this specific directory. Further discussion on this entitlement list may be found in Section 4.6.3. In the same way as the supernode, the dirnode stores a list of its children.

- Multiple **filenodes**, each representing a standard file. As well as for a dirnode, they hold an entitlement list. Alongside this, they contain the content of their associated file. Also, they are linked to their corresponding audit files.

4.3.2 Securing file's content

Each file's content must be encrypted to be secured from unauthorised parties. As encryption algorithm, we are using AES CTR. We chose a simple encryption algorithm instead of an authenticated one because we are not concerned about the file malleability. Indeed, as CTR mode of encryption doesn't produce an authentication tag, an attacker could easily tamper with the file. However, any modifications made by an attacker to the ciphertext will be easily spotted by a user (producing some weird content in the file). Nonetheless, we must cope with the core weakness of CTR mode of encryption: its nonce non-re-usability as can be demonstrated in Appendix E. To palliate to this issue, we must re-encrypt the file on each update. This re-encryption must each time use a new key/IV pair.

Re-encrypting the entire file on each update seems extreme and inefficient. To cope with that, we split the file into multiple blocks. Each block is encrypted with a different key. This way, we have efficient random file access. Indeed, when there is a file update, we just need to re-encrypt the edited blocks (e.g: when reading 1 byte of a 2GB file, we just need to decrypt the first block - similar with write operation) instead of the entire file.

To store these block encryption keys, we have built a metadata structure that we will be covered in Section 4.4.

4.3.3 Filesystem flow

Before processing *FUSE* requests, the Filesystem API must first do some prepossessing. This process follows the following steps:

1. Find the corresponding file's UUID: When mounted, the user sees the filenames in clear. However, they must be obfuscated before storing them on the remote storage. This means that to retrieve the correct file, we must first find the UUID corresponding to the given clear filename.

2. Check file existence: Once the UUID is known, we must check if the file still exists on the remote storage (in case it has been deleted by a malicious user).

3. Check the entitlement: Before processing *FUSE* request, we must first check if the current user can execute the *FUSE* action[1] on this specific file.

4.4 Metadata

As seen in previous sections, we need a data structure to store information corresponding to each filesystem node (e.g: file entitlement, file encryption keys and users registration). We will call this structure the metadata and will be covered in detail in this section. This structure will be secured thanks to the root key. As a quick reminder, it is the key that will be shared with other authorised users to access the filesystem.

4.4.1 Structure

As metadata are holding sensitive information about each node, it must be encrypted. However, some information is not required to be encrypted. This is why we designed the metadata to be composed of three parts:

- A **preamble section** holding non-sensitive information such as the size of the associated file for a Filenode, the UUIDs of its children for a Dirnode, etc. This section will only be secured with integrity protection. In this way, we are sure that this section can't be tampered with without being detected.

- A **secured section** holding the sensitive information that can't be divulged. It will thus be encrypted.

- A **cryptographic context** containing the information to securely store the other two sections. Such as the encryption key and the authentication tag

[1] Corresponds to a system call: READ, WRITE, UNLINK, GETATTR, etc.

to authenticate the preamble section. On top of that, this section will be authenticated and encrypted with the root key. In this way, this section will be at the same time tamper-evident and secured.

The idea is to have the metadata content secured by its cryptographic context, which in turn, will be secured thanks to the filesystem root key. The root key is protected by the Enclave itself. We applied this procedure to reduce the usage of the root key, and thereby increase its security as it is a long-lived key. The root key is set at the filesystem creation time and is never changed afterwards. Indeed, we can consider that the more we use a cryptographic key, the more it is susceptible to leak data that it was supposed to protect as it is just susceptible to more attacks.

4.4.2 Application to the nodes

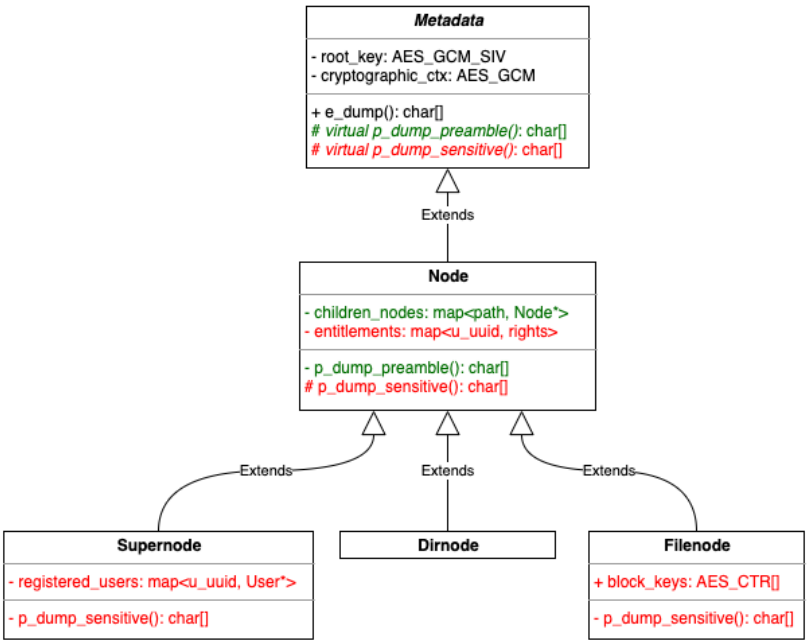


Figure 4.8: Metadata UML hierarchy

Now that we have seen the structure of a metadata, we can explain in details which data of a node is located in which section, as described in Figure 4.8.

First of all, we see that the metadata is an abstract class used as a wrapper to secure the preamble and secured sections, provided by children classes. In this way, children just need to return the plaintext content of their desired section and everything will be secured using the metadata wrapper.

Second, nodes are extending this abstract class through a generic Node class which contains information common to each node type: a list of child nodes and an entitlement list. The first list will be stored in the preamble section and the latter one in the sensitive section. We put the children nodes inside the preamble section because we don't care if an attacker can discover the structure of the hierarchy of the filesystem, as long as the directories and files names are obfuscated.

Last, each node type may store complementary information inside the sensitive section. According to the UML diagram in Figure 4.8, we see that the supernode stores the list of registered users and the filenode will store the list of block keys of its associated file.

4.4.3 Encryption procedure

As we have seen in the structure of a metadata, the cryptographic context is used to encrypt the required information. Here, we will more focus on the exact procedure and justify the encryption scheme we used.

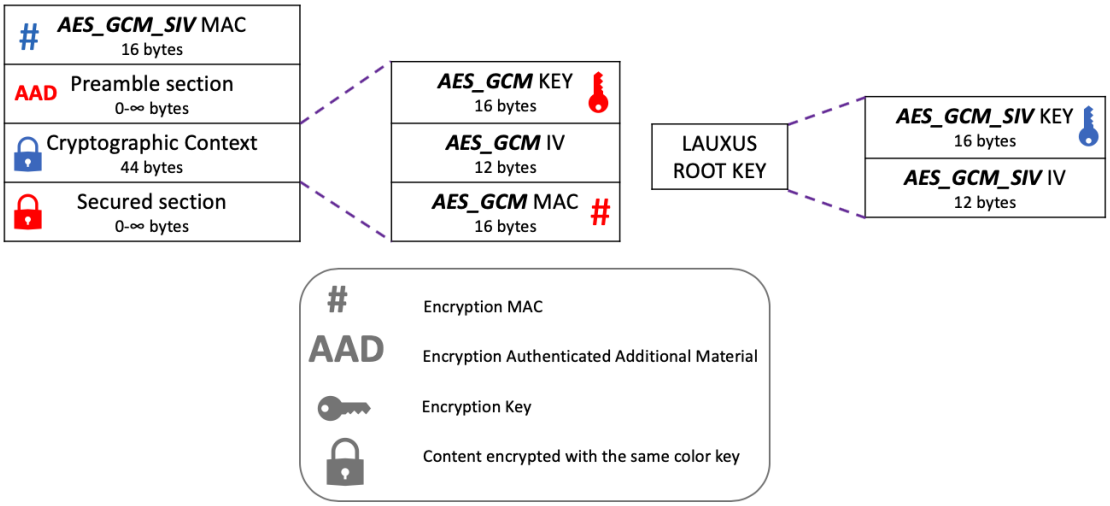


Figure 4.9: Metadata structure

This encryption procedure represented in Figure 4.9 explains the full structure of a metadata (as saved on the remote storage).

First of all, we start encrypting the secured section using an authenticated scheme (the red key protects the corresponding lock in Figure 4.9). The authenticated additional material used is the preamble section (the red **AAD** in Figure 4.9). This allows us to protect at the same time both sections: encrypt the secured section and produce a Message Authentication Code (MAC) linked with the preamble

section, making both ones tamper-evident. Afterwards, the resulting MAC will be stored inside the cryptographic context (the red # in Figure 4.9).

The authenticated scheme used is AES GCM which is widely used in today's current cryptographic system. As it is constructed on top of AES CTR, its main weakness is its nonce non-re-usability. Indeed, we can't encrypt multiple times using the same IV/Key pair even if the adversary doesn't know the IV, see Appendix E for demonstration. This justifies the update of the IV of the cryptographic context on every metadata content update.

Now that the metadata information is secured, we must secure the information used to secure the metadata content (which is the cryptographic context). As discussed above, we will encrypt them using the root key (the blue key protects the corresponding lock in Figure 4.9). For portability reason, the root key is composed of an encryption key and an IV. This allows us to use a single IV for all the metadata in the filesystem. This means that we can no longer use the AES GCM encryption algorithm as we are reusing the same IV. We will rely on a variant of AES GCM: AES GCM SIV (similar to AES GCM but with IV reusability). With this algorithm, we will encrypt the cryptographic context and save the generated MAC inside the metadata structure (the blue # in Figure 4.9).

4.5 Auditing Management

The idea of having an auditing file is to record the purpose of each critical action of the authorised users (reading or writing a file). The challenge is that each record (each purpose) of an auditing file must be protected and be self-standing (can be encrypted and decrypted without any other information than those present inside the record). Having this property, it also increases our performance and memory usage as we don't need to load information from untrusted memory into memory to encrypt an entry (which means we don't need to decrypt another content to encrypt an entry. Not like the metadata structure of a Filenode or a Dirnode).

Each auditing file is linked to a single file in the filesystem and is composed of multiple independent records.

To solve these challenges, we simply consider an audit entry to be like a metadata (cfr. Section 4.4). In this context, the metadata secured section is composed of: the purpose of the action, the time and the user who initiated the action. Each record is then chained requiring no additional encryption. Indeed, the cryptographic context of each record (metadata) is secured thanks to the audit root key.

4.6 User Management

Users within LAUXUS are identified by a unique random UUID and are authenticated with an asymmetric key pair. This key can be generated by LAUXUS if the user requested it. Asymmetric encryption enables secure authentication to the filesystem (e.g: like SSH authentication), as we will see later. We used Elliptic-curve cryptography keys as these primitives are, by default, implemented within SGX Enclave Package.

4.6.1 User registration/deletion

Users can only be registered by the administrator (cfr. Section 4.6.2). Users are added to the list of the authorised user simply by adding their public key and assigning them a UUID. The administrator is responsible for verifying the authenticity of the public key he received. The secure transmission of this public key is an orthogonal problem that will not be discussed here.

The user UUID will be used within the entitlement in Section 4.6.3. Afterwards, users can log in by using their UUID and their private key. We used random UUID to identity user instead of their "username" (like other classical login procedure). This chose is motivated by its simplicity and making more elaborated user management is not really in the scope of our work. Although, it would be interesting as an improvement to the project. Removing a user is even more straightforward, the administrator only needs to remove the user public key from the database.

During the above process (and also during the login procedure), the entire user database is loaded inside the Enclave memory which may incur issue as the memory inside the Enclave is limited (around 100MB shared between all the Enclaves on the computer). Discussion on ways to improve this issue is discussed in Section 5.3.

4.6.2 Login protocol

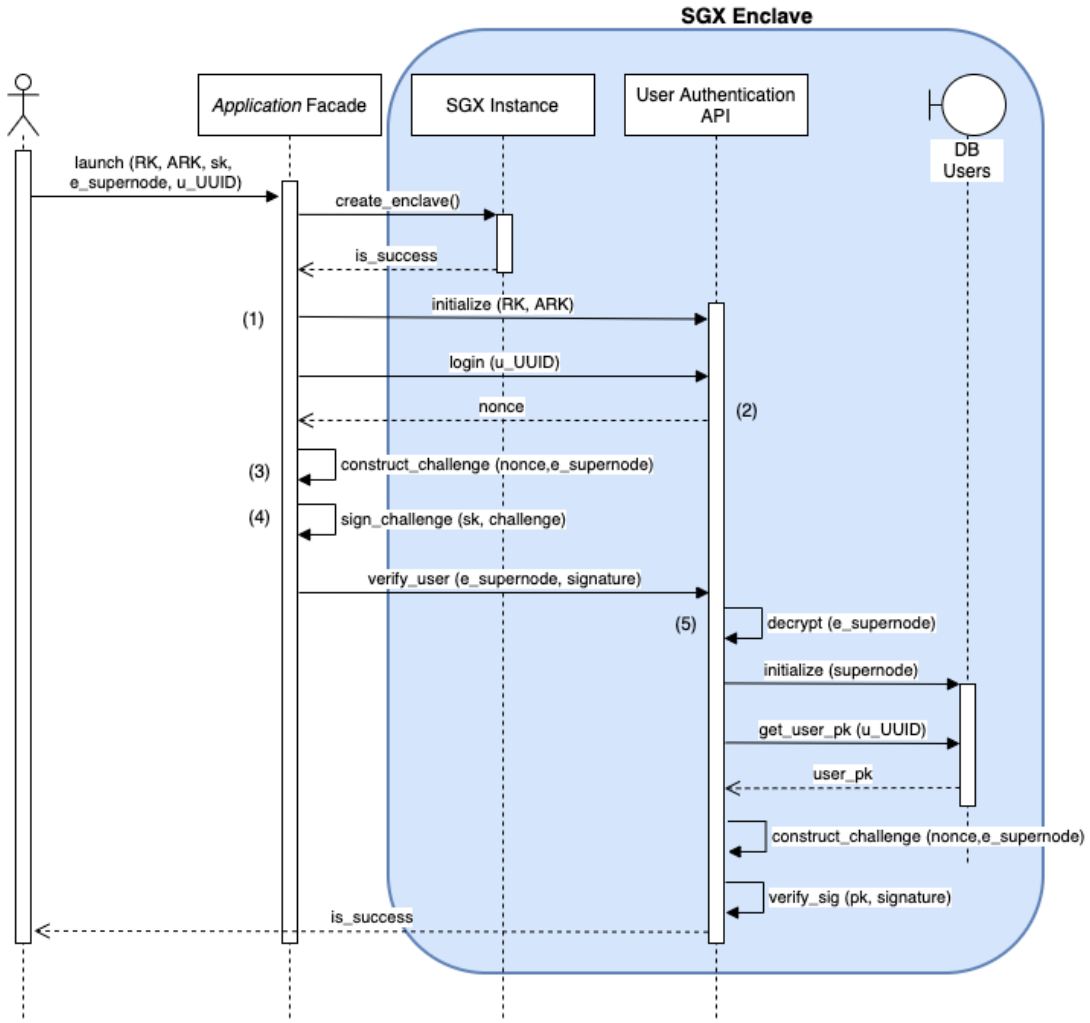


Figure 4.10: Login sequence diagram

The sequence diagram in Figure 4.10 describes the rather difficult login process. To help understand it, we will give complementary details (according to the numbers on the Figure):

1) *RK* and *ARK* stands for "Root Key" and "Audit Root Key".

2) The *nonce* is randomly generated inside the Enclave and is big enough to avoid duplicates (256 bit).

3) The challenge is constructed by concatenating the generated nonce and the encrypted content of the supernode. This allows to binds the nonce to the current state of the supernode. This avoids the supernode to be changed during the login procedure (in the case the user is revoked between the start of the login process and the end).

4) The signature and verification of the challenge is done thanks to the asymmetric property of ECC.

5) The supernode is decrypted following the protocol explained in the Metadata decryption Section 4.4.3.

In a nutshell, the login process allows to:

- **Verify the identity of the user** thanks to the signature provided and the public key stored inside the supernode.
- **The user is authorised to access the filesystem** thanks to the presence of his public key inside the supernode.
- **The supernode hasn't been tampered with** thanks to the metadata encryption protocol.
- **The supernode is linked to the given root-key** thanks to the metadata encryption protocol.

4.6.3 User entitlement

As we have discussed above, users are identified by using UUID. These UUIDs are used for the user entitlement. Each file and directory in the filesystem has an entitlement database stating which UUID can access the given resource. The entitlement is composed of 4 flags: ownership permission, read permission, write permissions and execution permission (1 bit per flag, like UNIX RWX permission). Only the administrator and the owner of a file can assign the entitlement to this specific file (a user is the owner of a file if the ownership flag is set to 1).

Note that these UUIDs are not related to the Unix user ID. LAUXUS never rely on these Unix user ID. The only exception is when providing the UNIX owner of a file (upon **getattr** system call). We decided that the user who launched LAUXUS will be the UNIX owner of all the files he can access. For security reason, only the UNIX owner of the file can access it. Other UNIX users (users in the same group or others) permissions are always set to 0. In this way, no-one except the launcher of the application (the user we authenticated) can access the filesystem.

4.7 Sharing the Filesystem

Now that we have seen how LAUXUS works on a local computer, we can dig on the sharing capabilities. As we have seen, the root key is the key material for accessing the Filesystem. Currently, we know that it is sealed on the owner computer. This allows binding the root key to his computer. Indeed, once a material is sealed with SGX, the only party that can unseal it is the SGX instance that sealed it in the first place. This means that we can't just share the sealed version of the root key to other users. As we depicted in Figure 4.3, we need an out of band channel for transmitting the encrypted version of the root key. This is what we will cover in this Section.

Before going further, let's remember that currently, every user owns an asymmetric ECC key pair allowing other users to authenticates a message sent by another users (we consider that all the public keys are securely accessible for all the users). This functionality will become handy in our protocol.

The sharing protocol is based on the ECDH (Elliptic Curve Diffie-Hellman) particularity. ECDH enables to compute a secret from two key-pairs. Mathematically, if we have two key-pairs: (pk_1, sk_1) and (pk_2, sk_2), a shared secret K can be computed in the following way:

$$K = ECDH\_SECRET(pk_1, sk_2) = ECDH\_SECRET(pk_2, sk_1) \tag{4.1}$$

With *ECDH_SECRET* being a known algorithm in the Elliptic Curve domain.

In practice, this means that: two users can securely compute the same secret just by knowing the public key of the other user.

As terminology, we will use the same users as in Figure 4.1. This means that Alice will share the root key with Carl. In a nutshell, the protocol will be split into 4 phases:

1. Alice and Carl will create a message attesting that they are using a valid SGX Enclave. This message will also bind the user to its machine. This message will be called: the *authenticity message*. The *authenticity message* is simply a quote generated by the SGX Enclave in which we add additional information.

2. Alice will check the authenticity and the validity of the *authenticity message* created by Carl.

3. Alice will derive a shared secret (this secret is bind to this root key transaction) using the Equation 4.1. She will then encrypt the root key with this secret and share the result with Carl.

4. Carl will check the authenticity and the validity of the *authenticity message* created by Alice.

5. Carl will derive the same shared secret as Alice and will then be able to decrypt the root key before sealing it on his computer.

What this all means is that: Alice will encrypt the root key that only Carl can decrypt. This encryption key is unique and binds both users and both user's computers. This means that every time Carl is using a different computer, the protocol must be re-run.

To avoid using a third-party server, the above messages will be stored on the remote storage. We this in mind, we created a hidden folder inside the Filesystem containing one directory per user. Each user directory will contain the different message required to add the specific user to the Filesystem.

Phase 1 - Creating the *authenticity message*

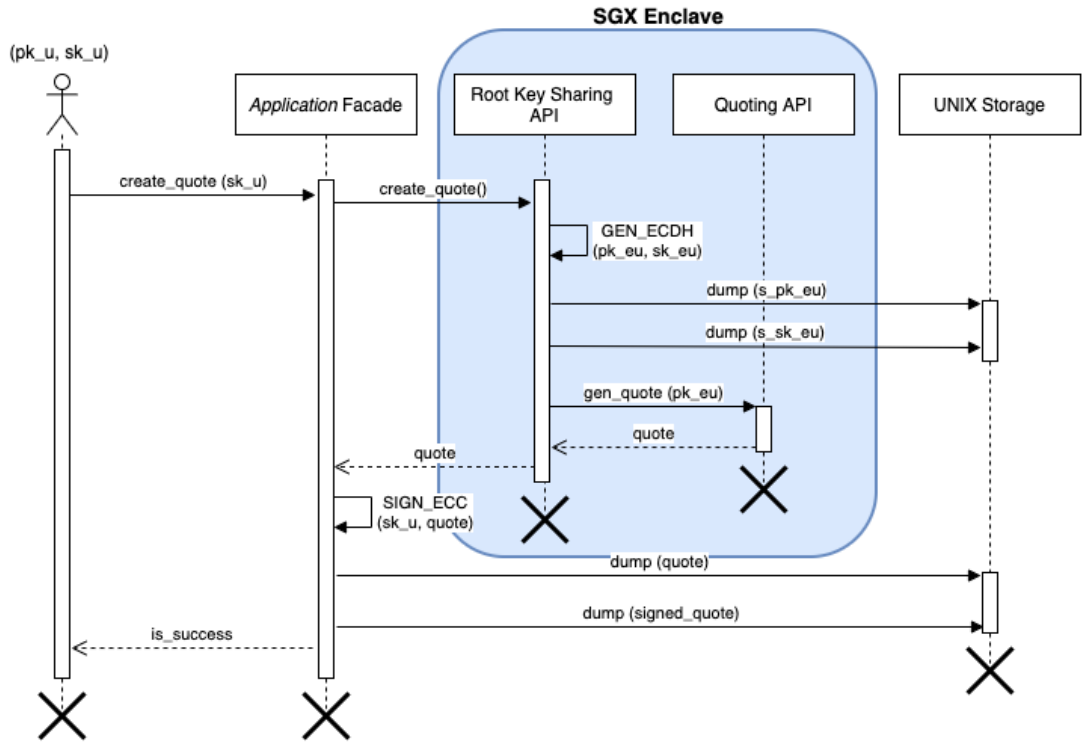


Figure 4.11: Phase 1 Sequence Diagram

The Figure 4.11 explains the first Phase and is based on the Quoting capability of SGX Enclaves (cfr. Chapter 3). To bind the user to his computer, we create

another key-pair[2] inside the Enclave and seal them. The quote bears the Enclave public key and the quote is signed using the user private key for authentication.

Phase 2 - Verifying the *authenticity message*

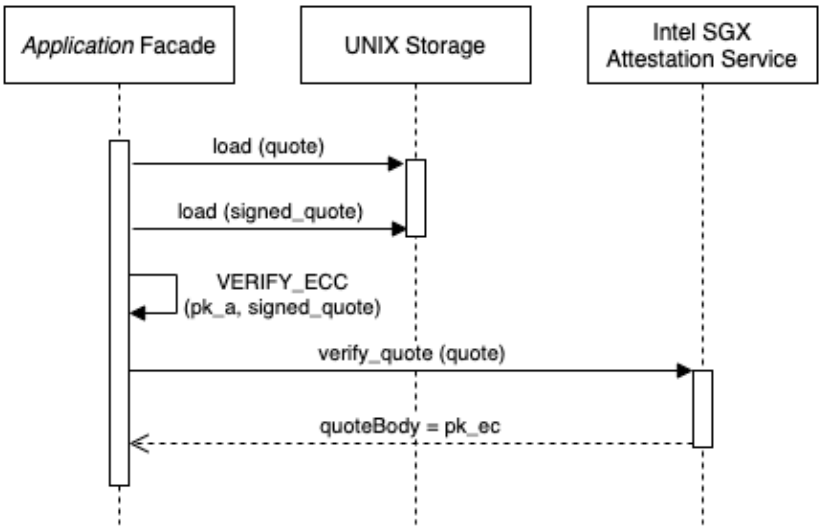


Figure 4.12: Phase 2 Sequence Diagram

The Figure 4.12 explains the second Phase and is based on the remote Intel Attestation Service furnished by Intel (cfr. Chapter 3). Upon success, the initiator will be sure that this specific user is using a valid SGX Enclave.

[2]pk_eu stands for the public key of the user U's Enclave - similar notation with the secret key.

Phase 3 - Encrypting the root key for sharing

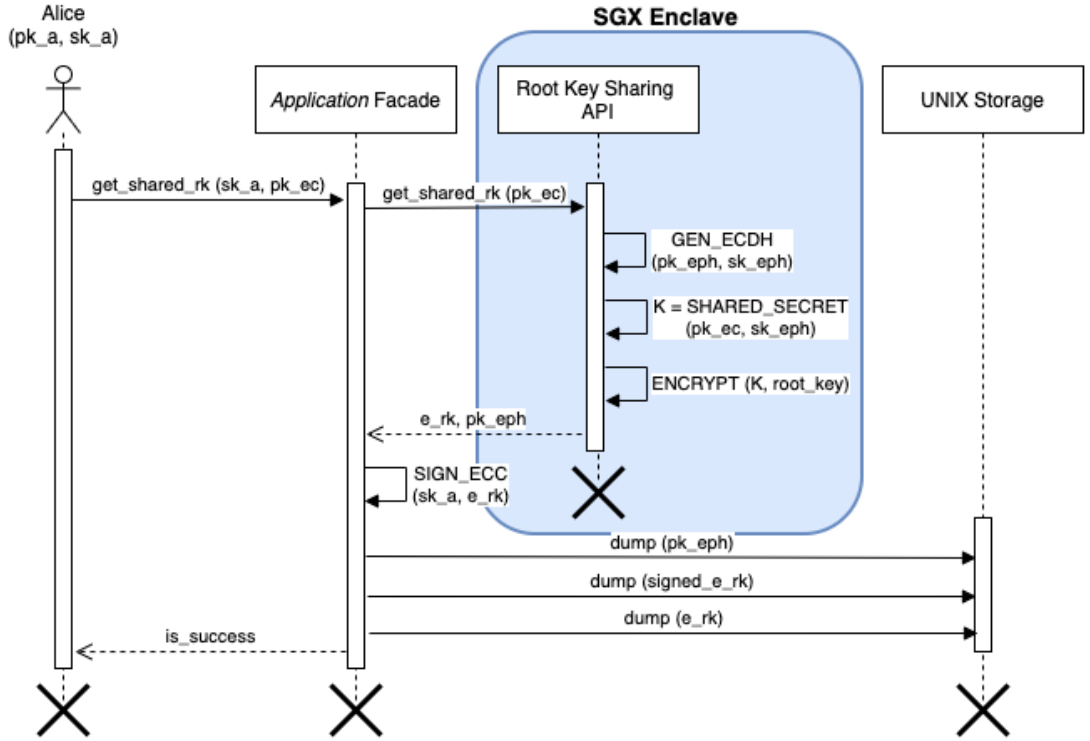


Figure 4.13: Phase 3 Sequence Diagram

The Figure 4.13 explains the third Phase which is made by the administrator one time for each other user (if the other user never changes his computer). As explained in the Sequence Diagram, Alice creates an ephemeral key-pair[3]. The encrypted root key is signed with Alice private key to testify authenticity.

Phase 4 - Decrypting the shared root key

The Figure 4.14 explains the fourth and final Phase which is straightforward knowing the third Phase. We just need to do the opposite operation. The shared secret reconstructed thanks to the ephemeral public key.

[3]pk_eph stands for ephemeral key-pair - similar notation with the secret key

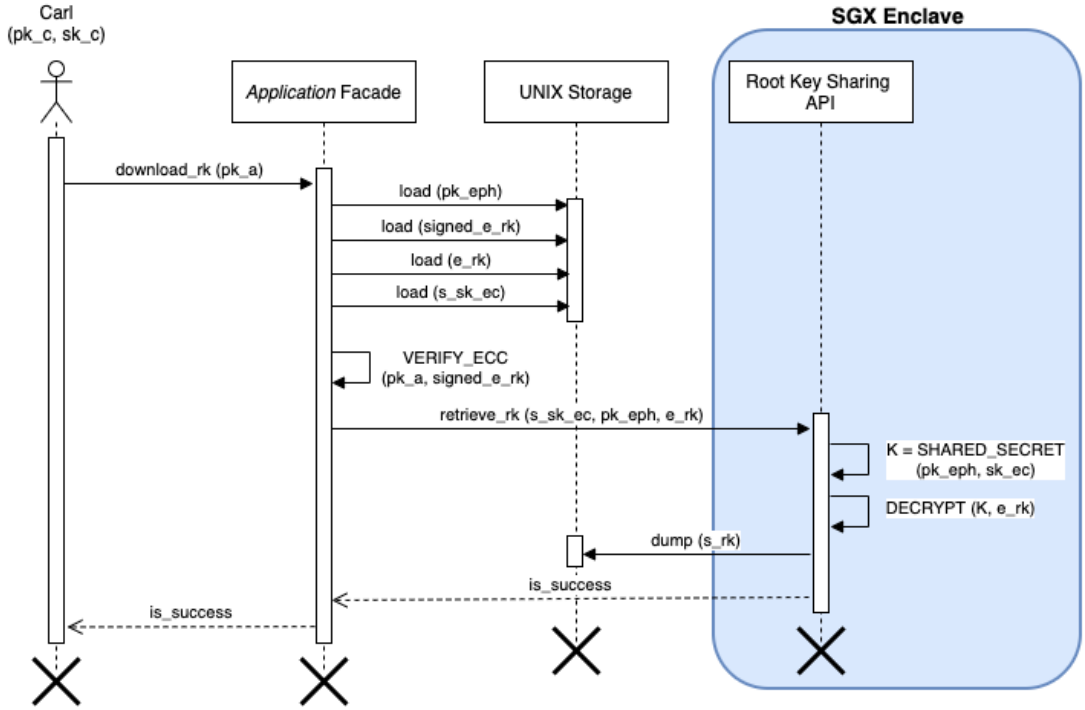


Figure 4.14: Phase 4 Sequence Diagram

4.8 Additional information

Additional information about LAUXUS implementation can be found in the Appendixes. Such as:

- LAUXUS Life State Diagram explaining the state diagram of each user based on his role (cfr. Appendix A).
- LAUXUS Login Overall process explaining the detailed step by step procedure that must be followed for an administrator to add a new user to the Filesystem (cfr. Appendix B).
- LAUXUS Structure in action which shows what LAUXUS will look like in real (the difference between the view of the authorised user and the one seen by the remote storage). This show how a single file is split into three subsequent files (one for the content, one for the audit and one for the metadata)(cfr. Appendix C).

Chapter 5

Analysis

5.1 Security analysis

The below Figure 5.1 shows the security assessment of LAUXUS based on the theoretical description seen in Chapter 4. We based our table on the STRIDE model as explained in the problem statement. As we can see from the table, we aren't protected against all the listed threats. This is why we added an improvement column to explain what can be done to mitigate the concerning threat in a future LAUXUS version. As some of the threats and remediations are straightforward to understand, we will only focus on the more complex ones as per respect to their index:

(1) As audit entries are self standing (cfr. Section 4.5), there is no linkage between the entries inside the audit file. Which means that if an attacker can precisely remove a whole entry, nobody will be able to note the modification. Note that this is not an easy task for the lambda attacker as the entries are not fixed size (depends on the length of the purpose), however, if the attacker knows the length of the original purpose, it becomes trivial. Multiple remediation[1] are possible but the most efficient one is to include the number of audit entry inside the metadata structure. This allows keeping an excellent audit entry write overhead while solving the issue. Indeed, the whole metadata structure is loaded and written anyway from/to disk.

(2) In the same manner as the attack **(1)**, an attacker could replace a whole entry with another one (e.g: generated from another LAUXUS instance). Even if it seems feasible, we must not forget that each audit entry is protected thanks to the audit root key. As root keys are unique to each LAUXUS instance, it becomes impossible for an attacker to forge a valid audit entry.

[1] Store audit keys inside the metadata, Hash of the whole plain file, Link list of audit entries.

	Index	Threats	Remediations	Improvements
Spoofing		▪ Steal valid user's credentials (private key) ▪ Replacing user credentials written inside the Supernode ▪ Malware on the authorised user computer	▪ / ▪ Secure Authenticated Encryption ▪ Information are either store encrypted on the untrusted storage or inside SGX	▪ Up to the end-user to secure its credentials ▪ / ▪ /
Tampering		▪ Updating data file	▪ Content encrypted (no MAC) -> Garbage decryption	▪ Use Secure Authenticated Encryption
Repudiation	▪ **2** ▪ **1** ▪ **1** ▪ **3** ▪ **4**	▪ Edit the audit file to alter an entry ▪ Replace whole entry of an audit file ▪ Remove whole entry of an audit file in order to hide a given action ▪ Deleting the audit file in order to hide all actions done to a file ▪ Renaming the audit file in order to swap the audit entries with another audit file ▪ Replace a whole entry from another entry in the same FS	▪ Secure Authenticated Encryption ▪ Audit entry protected with Root Key ▪ / ▪ / ▪ / ▪ /	▪ / ▪ / ▪ Counter of audit entry inside the metadata structure ▪ Counter of audit entry inside the metadata ▪ Link audit file to the concerned file ▪ Link each entry to the concerned file
Information disclosure	▪ **6** ▪ **5**	▪ Reading the data file ▪ Authorised users disclosing secret encryption keys ▪ Authorised users disclosing a file content	▪ Content encrypted ▪ Encryption keys are kept inside SGX Enclave ▪ /	▪ / ▪ / ▪ Minimize actions (e.g: copy outside FS, etc)
Elevation of Privilege	▪ **7** ▪ **8** ▪ **9** ▪ **10**	▪ User entitlement edition ▪ Steals Root Key in order to access the Filesystem ▪ Switching two saved files content in order to access unauthorised content ▪ Switch parent directory in order to access the file content ▪ Using a previous version of the Supernode in which the attacker had access ▪ Using previous version of a Filenode content/metadata in which the attacker had access ▪ Replace Root Keys to access the Filesystem	▪ Secure Authenticated Encryption ▪ Root Key sealed inside SGX Enclave ▪ Content encrypted (no MAC) -> Garbage decryption ▪ MAC on the FS hierarchy ▪ / ▪ Block encryption keys will be incorrect ▪ Root Keys are sealed thanks to SGX	▪ / ▪ / ▪ Use Secure Authenticated Encryption ▪ / ▪ Use version number in the Supernode ▪ Use Secure Authenticated Encryption ▪ /

Figure 5.1: Security assessment

(3) An attacker could rename an audit file to another one to swap them. Currently, our implementation is vulnerable to this attack. Indeed, inside the audit file, there is no link to the data file. To solve this, we just need to bind the two files together. An easy way to do so is to add a small section at the beginning of each audit file. This section contains the name or the UUID of the concerned file (if filename used, we have an issue when renaming the file). This section must be encrypted using the key stored in the cryptographic context of the corresponding metadata structure. We can't use the root key as the encryption key, as an attacker would only need to swap the section with another audit file. This process is very efficient as the new section must only be written once (at the corresponding file creation).

(4) Similarly to the attack **(3)**, the only way to solve this issue is to include the UUID of the concerned file inside each audit entry (using the filename would result in complications when the file is renamed).

(5) As discussed in the threat model, once an authorised user can access a certain version of the Filesystem, we can't do anything to protect this version if the user chooses to make this information public. However, if we can control the OS or the user-space application[2], we can prevent the user from copying the decrypted content outside the system. This will prevent the user from sharing it. However, it can only be done to a certain extent, the user can still take a picture of the decrypted content.

(6) Similarly to the point **(5)**, a malicious authorised user may wish to share the encryption keys with another unauthorised user. In a classical scenario, this would be impossible to prevent. Fortunately, thanks to SGX secure computation, we hide all the encryption keys to the user, preventing them from disclosing them.

(7) To access the content of a file an authorised user hasn't access to, the user may want to switch the file content with another file to which he has access. In a nutshell, given two files A and B in which user X can access file A, user X may want to access file B. The idea is to switch the content of file A and B but keeping the metadata structure of file A (and thus the user entitlement). With our implementation, nothing will be noticed if the two files have the same number of blocks or file B has more blocks than file A. Indeed, the block keys are stored inside the metadata structure, so LAUXUS will just read a file block and decrypt it using the corresponding block key. However, as the key is incorrect, the content will be decrypted into garbage

[2]Something like Digital Rights Management (DRM) features (Cfr. [31]).

but without throwing any error. To improve this, we can use an authenticated encryption algorithm to throw an error. Using an authenticated algorithm will also enable the possibility to use forensic (warns the administrator of inappropriate behaviour). Furthermore, with this forensic, we know on which computer the inappropriate action has been done.

(8) Similarly to the attack **(7)**, an attacker may change the parent directory of a file to one that he is allowed to access. This process is impossible thanks to the authenticated encryption on the metadata structure. Indeed, even if the parent directory is in plaintext (in fact, it is the children nodes that are in plaintext), the information is protected as it is passed as the Authenticated Additional Data inside the GCM encryption algorithm.

(9) An attacker could use a previous version of the supernode while keeping the rest of the Filesystem. A version in which he was allowed to access the filesystem. Our implementation doesn't cope with this threat however a solution is possible. We can use a simple version number on the Supernode. Each time the Supernode is updated, this version number increases. However this is not enough, we must also link the Filenodes and Dirnodes to this version number. Consequently, each node will also have a version number. However, the node version number will only be updated with the one of the Supernode whenever the node is accessed. Furthermore, a user can only access a node if the node's version is smaller or equals to the Supernode version. This process may seem imperfect because nodes version are only updated when they are accessed. Let's demonstrate this with a simple scenario:

We consider Alice and Bob, both can access version 1 of the Supernode. Alice decides to revoke Bob creating a new Supernode with version 2. At this time, Bob can still access the filesystem and all the files he used to access by simply using the previous version of the Supernode. This is not a security issue because Bob had already access to these files. When Alice chooses to edit a file X, it will assign a new version to this file. Now, Bob can no longer do the same trick and don't know the new version of the file because its Supernode version is lower than the new version of the file. This scenario proves the effectiveness of our process.

(10) Similarly to the attack **(9)**, an attacker may use the previous version of the metadata or the content of a file. This idea behind this action is to use the old user entitlement (where he had access to the file) and combine it with the new version of the content (which he should not have access to because he has been revoked). The protection is similar to the one seen in the attack **(6)**.

After analysis, we can conclude that: once a user has access to a version X of a file, he will always be able to access this version of the file (through the different attacks seen above). Furthermore, in the case the end-user computer is hacked, the attacker can only access the content that the targeted user can access, which may be very small. This means that our model is relatively strong against a Man-In-The-Computer attacker.

As we have seen from the above description, although our theoretical model is quite complicated, it still has some flaws. However, these flaws are not so complicated to fix and can be implemented in a relatively small amount of time compared to writing the entire code base. The flexibility of our codebase makes these tasks even easier.

5.2 Practical performance

In this section, we will look at the performance of our implementation. To do so, we developed a benchmark to showcase the performance of multiple scenarios. In each of these scenarios, we start with an empty filesystem to have a common base between all the situations. The benchmarks are run inside an Intel NUC running Ubuntu 18.04.4 with no particular intensive process already running. Furthermore, tests are run using the latest SGX SDK[3]. To avoid biased results, each test scenario is run up to 100 times.

During all the below scenarios, we will compare three filesystems: LAUXUS, the Linux ext4 and the *FUSE* passthrough. This last filesystem just mirrors the usual filesystem (ext4). We chose to compare LAUXUS to ext4 to see the overhead incurred to the standard filesystem that most end-user is probably using. The comparison with the *FUSE* passthrough was to see the impact of using *FUSE*. The strange behaviour that we observed is that the *FUSE* passthrough is more efficient than ext4 which is pretty particular but won't be discussed here as it is not the main concern of the work. We just want to compare LAUXUS performance compared to other filesystems.

Copying operation overhead

In this experiment, we will look at the performance of writing a file into LAUXUS. This aims to see how our filesystem performs and deals with writing file of different sizes[4]. Note that we are more interested in the evolution of the time instead of its absolute value. Indeed, having an exponential evolution is a huge limitation.

[3]SGX 2.9.1 at the time of writing.

[4]we only checked the performance with the write operation because the read operation is similar and provides the same results.

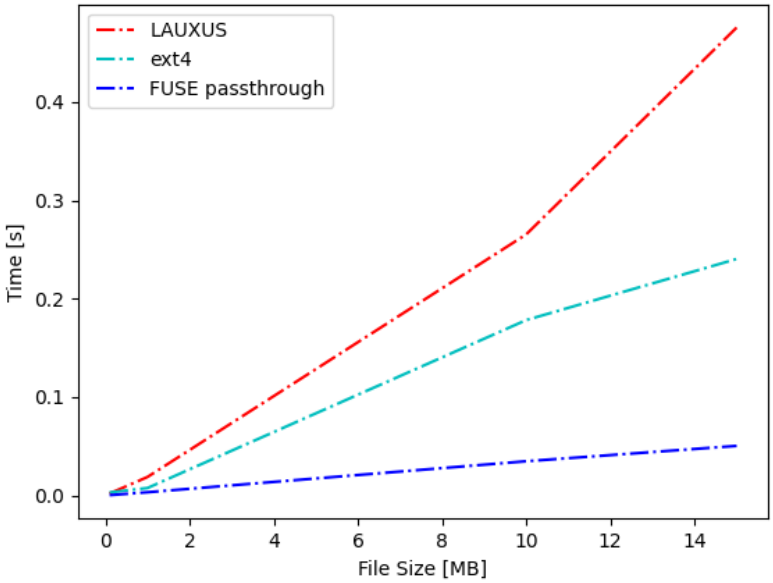


Figure 5.2: Copying time VS Size of the file

From the design description, it is expected to have a linear evolution because, upon write, we only need to encrypt the edited block (which means it doesn't depend on the file size) and not the entire file.

As we can see from Figure 5.2, our implementation fits with our model because of the nearly linear evolution. Besides, we see that we have a sizeable overhead (two times slower than with ext4). Upon closer look, we see that the curve is not absolutely linear. This is because the size of the metadata structure increases throughout the copy operation. Further explanation of this behaviour is discussed in a subsequent section.

Block writing time throughout an entire file write

In this experiment, we aim to see the evolution of the time it takes to write a block during a copy operation. We would expect from our model to take roughly the same amount of time to write each block during a copy operation.

As we can see from Figure 5.3, the write time is not constant throughout a full copy operation. This explains why we don't have a perfectly linear evolution in Figure 5.2. The reason for this behaviour is that on every write operation (write a single block), we are writing the entire metadata structure to the remote storage. We need to do so as we are creating a new content block and thus we must create a

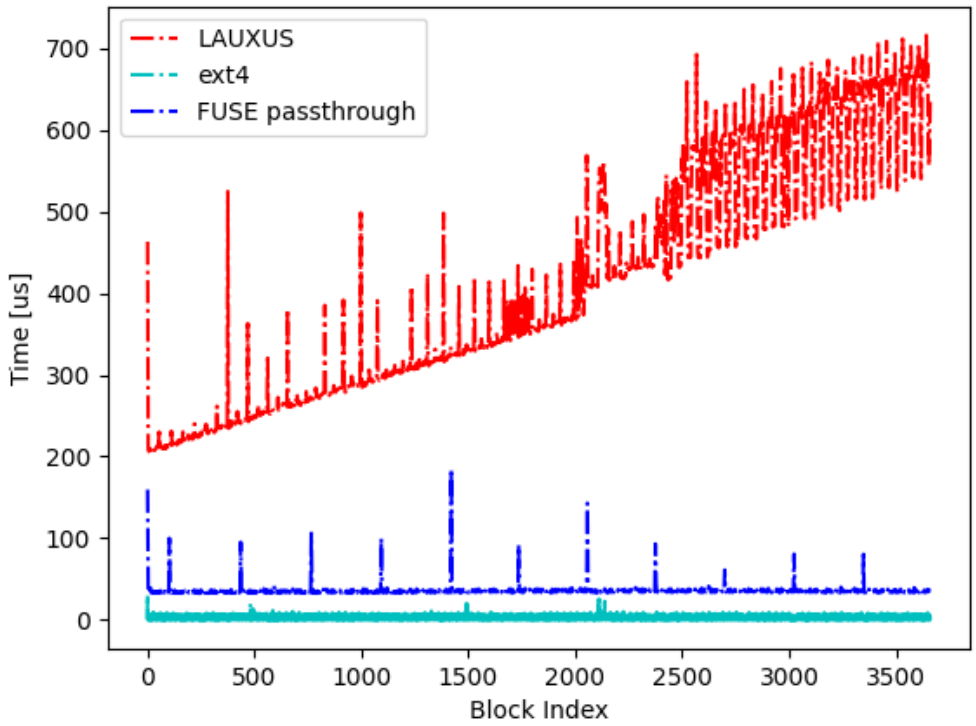


Figure 5.3: Evolution of the block writing time for a 15MB File

new block key. As the metadata structure becomes bigger and bigger, it takes more and more time to encrypt and write it. This is why we don't have a constant time.

User entitlement overhead

This test is important because we have user entitlement on each directory level. Furthermore, because the names of the files are obfuscated, we must load the parent directories of the concerned file. Indeed, each Dirnode contains a mapping between each of its children name and their corresponding UUID. Furthermore, as there are multiple Dirnode level (with each a possibly different user entitlement), to de-obfuscate a file, we must load all of its parent directories.

This means that theoretically, the deepest in the hierarchy the file is (number of parent directories to the root), the more we need to check the user entitlement and load nodes, increasing the overhead on the access operation. Which means, we expect a linear evolution compared to the depth of the file. Fortunately, this process is done only one time: when the file is opened. Once open, the file structure

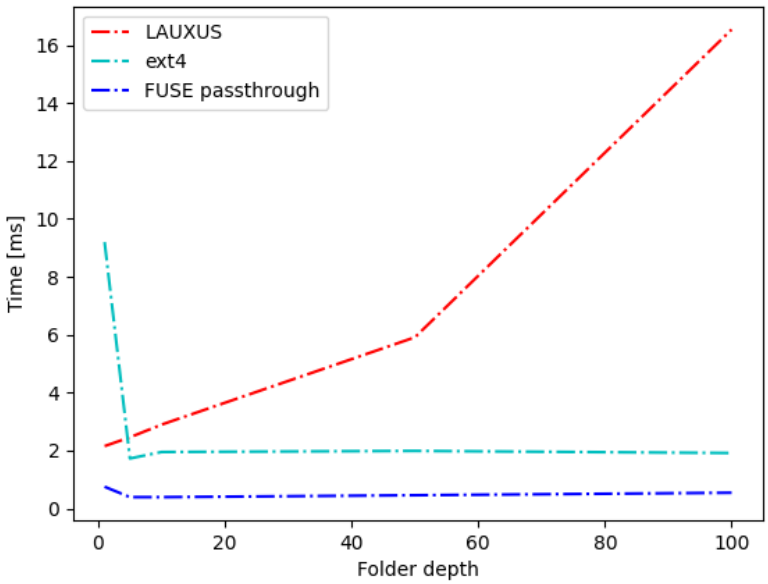


Figure 5.4: Copying time VS Depth of the file

stays in memory and we no longer need to check the hierarchy nor to load the parent directories to find the Filenode's UUID.

The Figure 5.4 illustrate the above theoretical explanation.

We see that this cause a considerable overhead when there are a lot of directories. However, increasing the delay to access a file of a few milliseconds is hardly noticeable to the end-users. As we are not aiming to use our filesystem in environments where extremely low latencies are required, we don't think that this requires further improvements.

Writing a few bytes overhead

The idea behind this test is to prove that if we need to make a small modification to a big file, it has the same performance as making the same modification to a way smaller file. Indeed, as our model is splitting a file into multiple blocks, when we edit a block, we just need to encrypt and save this precise block, not all the others. This means that we should expect no evolution when the file becomes bigger.

By inspecting Figure 5.5, we may think that our implementation doesn't fit our model as our time evolves kind of linearly. It is not constant because we must first load the metadata of the filenode in memory. This simply means that the bigger the file is, the bigger the metadata is and thus the longer it takes. We can

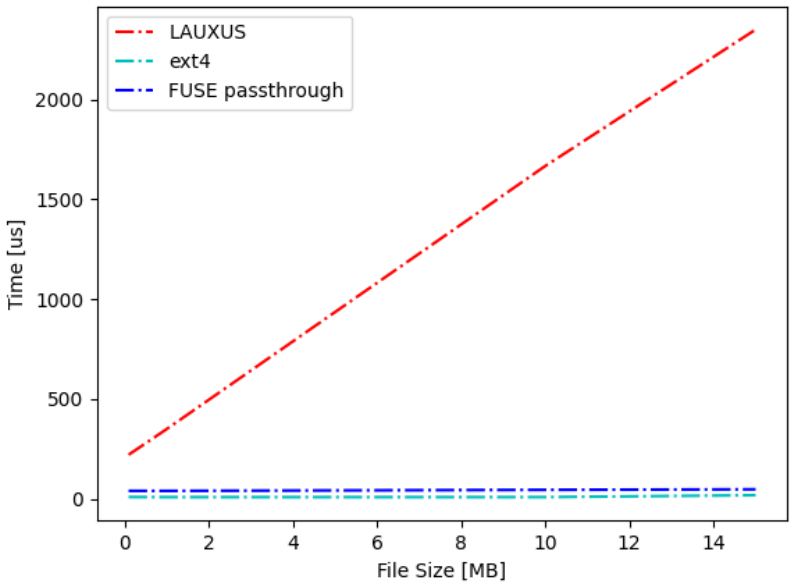


Figure 5.5: Writing a few bytes at different offset inside a 15MB file

be pretty confident that this reasoning is correct because, between the smallest and biggest file (from 0.1 MB to 15 MB), it increases of only a few milliseconds.

Block size impact on copy operation

Depending on each user-space application, the application may write their information with blocks of different sizes (e.g: nano makes write operations of maximum 1KB, vim makes write operations of maximum 2KB, etc). This test is to check how our implementation behaves in those cases. Having a non-constant time means that depending on the user-space application the end-user is using, it may result in different performances.

By inspecting Figure 5.6, we see that we are more efficient if we are writing blocks of larger sizes.

As our implementation is using a stateless model (no session) for writing the encrypted file to the remote storage, the more we increase the block size, the more we are efficient (to a certain threshold). By stateless model, we just mean that on each write operation, we open the file on the remote storage then apply the operation then we close the file. A more efficient approach would be to open the file at the same time the end-user opens it (with the open operation) and similarly when closing it. However, this should not highly change our performances as the

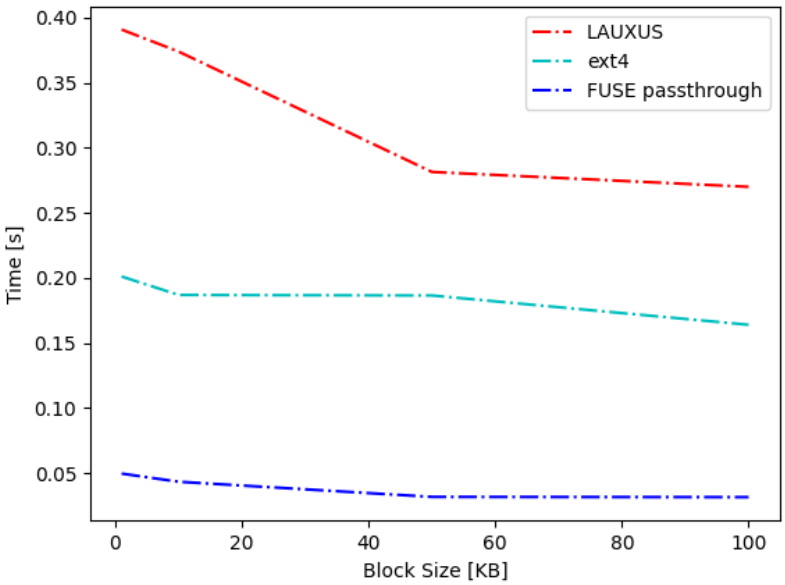


Figure 5.6: A 10MB File writing time VS block size

writing time is mainly governed by the encryption time.

Unzip folders

This scenario is to showcase that LAUXUS can be used by other filesystem programs (zip in this case). The scenario is really simple, it consists of copying an archive inside LAUXUS, unzip it, list all the files in the unzipped folder[5] and lastly delete the entire unzipped folder. We chose this scenario as it regroups a lot of different operation that could be done in a real-life situation. The chosen archive is simply the LAUXUS repository. It has a size of 1.4MB zipped and 3.2MB unzipped.

As expected, LAUXUS is slower than both filesystem but in the end, the whole operation is still pretty fast and doesn't incur a noticeable overhead to the end-user.

Surprisingly by analysing the Figure 5.7, we see that ext4 becomes faster than the *FUSE* passthrough which is the opposite than what we saw in the other scenarios. However, this seems logical as *FUSE* can't be faster than ext4 because it just mirrors ext4. We suppose that the reason the other scenarios were at the advantage of *FUSE* may come from a special operation in *FUSE* to regroup operation together for efficiency.

[5]using the tree command.

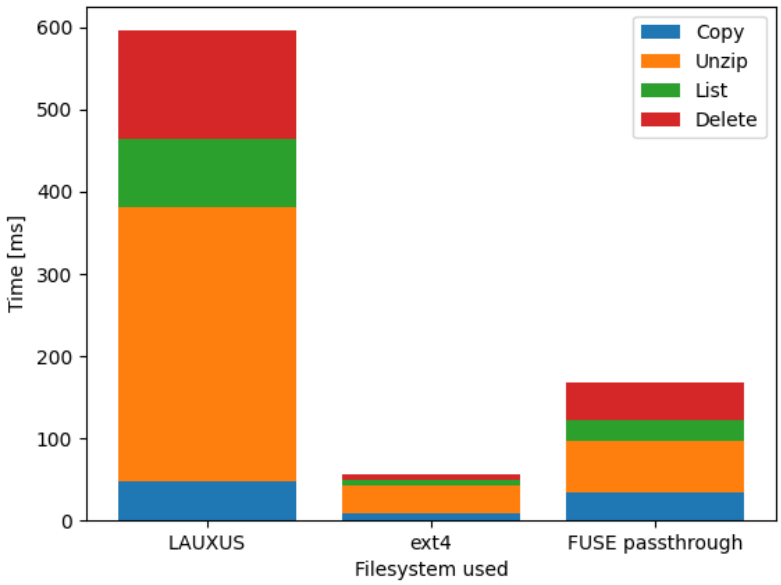


Figure 5.7: Zip scenario

Memory and CPU usage

This last test is rather different than all the above ones. We don't compare to the other filesystem (ext4 and passthrough) because we are more interested in the absolute values and how they evolve. Furthermore, the memory usage is specific to the Enclave. Indeed, this measure represents the maximum amount of memory needed to execute the operation. The operation chosen here is just copying a single file like in Section 5.2.

By analysing the Figure 5.8, we see that our software is quite demanding in memory space as discussed in Section 5.3. Indeed, as inside an Enclave, both the Stack and Heap size are limited, we can encounter issues when dealing with a very big file (e.g: 1GB). Solutions for this limitation are explained in 5.3.

The good news about all the curves is that they don't behave exponentially which is very promising for the software. This means that if the issues discussed in Section 5.3 are solved, the application can be used without limitations (e.g: with very big files, etc).

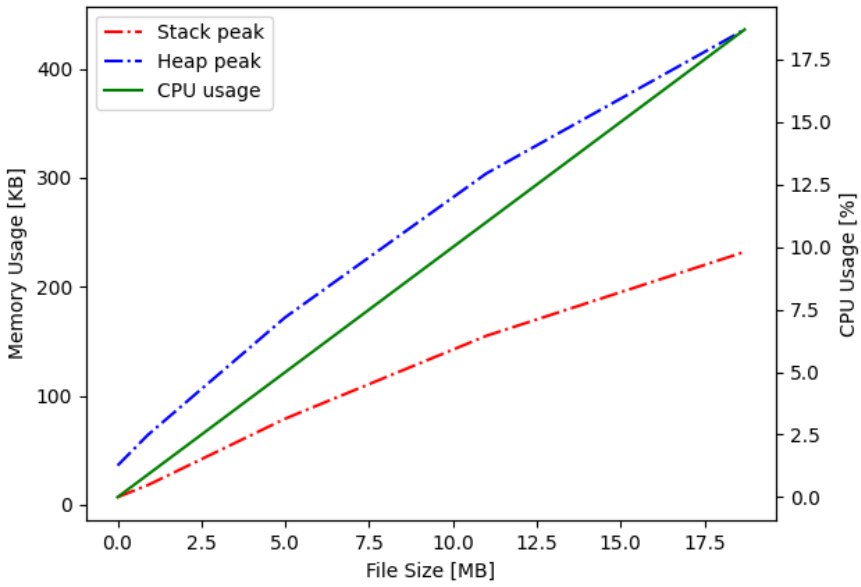


Figure 5.8: Memory and CPU usage when copying a single file

5.3 Discussion

Now that we have covered the security of LAUXUS, we can look at its limitations. As we have already covered the security limitations in the previous section, we will focus more on the functionalities limitations.

- **File size:** There is a maximum size to the files that LAUXUS can manage. This limitation is caused by the fact that we load the entire Filenode metadata structure in memory (more precisely on the Heap). As the file becomes larger, it is composed of more blocks and thus of more block keys stored inside the metadata structure[6]. Of course, this problem can be temporally solved by either increasing the Heap size or increasing the block size (which is by default to 4KB). A more permanent solution would be to use pagination on the block keys of the metadata structure.

- **User database size:** Following the same reasoning as above, we see that we are limited in the number of users registered inside our Filesystem. Furthermore, the supernode stays in memory during the whole time the Filesystem is running. This means that it also limits the size of the files we can load (Cfr. explanation above). To cope with this we can first discard from memory the

[6] 20 MB File with 4KB block -> 5000 block keys of 32B -> metadata structure of 160KB.

user database once the user is logged. Second, we can also use pagination inside the users' database.

- **UUID generation:** There may be a collision when generating the UUIDs for the users or the files. Indeed, all the UUIDs are randomly generated. However, each UUID is composed of 64 bits. This means that by following the birthday paradox, we have a 50% chance of having a collision after generating 2^{32} UUIDs. Theoretically, it may not be the best idea but in practice, we will never encounter a collision as there are just too many permutations and not enough UUID generated.

- **Prompt action purpose:** Currently, there is no nice GUI to ask the user to provide the purpose of his action. Despite long research, I wasn't able to find a way to ask for the user purpose. As it was not the primary goal of this work (the goal was more to prove that the concept was feasible) I chose to let the purpose to be a static string. Once someone finds out how to ask for user input from LAUXUS, everything will be good to go as the backbone of handling user purpose is already implemented in the Application Facade. Another alternative might be to provide a purpose when we login into the Filesystem (the purpose stays the same throughout a "session").

- **Most important syscall implemented:** LAUXUS implements the most important syscall[7] operation to have a working Filesystem. We didn't implement all the *FUSE* system calls because most of them were out of the scope of our work[8]. As the default behaviour of *FUSE* is to through an error code when a call is not implemented (instead of acting as a pass-through), some program may not work (e.g: such as Git).

- **Persistence of the user entitlement:** Currently, when a user is revoked from the Filesystem, only the Supernode user database is updated, not the user entitlement of the files. We chose to do it this way because if we had to load and update all the nodes of the Filesystem when a user is revoked, we would incur a huge overhead to the revocation procedure which should be fast. As the user entitlement works with the user UUID, a new user might inherit the user entitlement of the previous user with the same UUID. However, as discussed above, this probability of a UUID collision is extremely low thus doesn't incur an important security threat. A way to decrease even further this "threat", is to write a small script that he is run before mounting LAUXUS that check the Filesystem structure and along the way correct the

[7] open, create file/dir, read, write, release, truncate, delete file/dir, readdir, getattr
[8] symlink, flush, link, chmod, chown, sync

small incoherence (such as the when there is the UUID of a revoked user inside a file user entitlement).

- **Concurrent writing operation:** Currently, LAUXUS will end up in an incorrect state when there are concurrent updates on the same file (e.g: two users edit the same filesystem when offline and then upload them on the remote storage). Indeed, as we must update the IV of the metadata structure on each file update (cfr. Section 4.4.3), if two users edit the same file at the same time, it will result in two different IV and classical merging algorithm won't be able to resolve this. The simplest solution to this is to use AES GCM SIV instead of AES GCM due to its nonce re-usability (cfr. Section 4.4.3).

5.4 Improvements

Last, next to previous sections, we will look at the improvements that can still be brought to LAUXUS to make it more efficient and more user-friendly.

- **Inherit user entitlement:** This improvement will ease the work of the owner user when assigning new rights to an authorised user. The idea is that a Node inherits the user entitlement of its parent Node. This would mean that if an authorised user has a read right on a Dirnode, he will have a read right on all the Filenode inside the specific Dirnode. Of course, this inheritance can be overridden in any way the owner user wishes (e.g: allowing the authorised user to write a specific file in the above Dirnode).

- **On-demand audit files:** Currently, an audit file is automatically created for each file created on the Filesystem (along with the corresponding purpose prompt). This can be rather annoying because of temporary files (e.g: when we open a file in vim, vim creates a *.swp* file to store his historic of changes and other information). Imagine if the user must provide the purpose of the action every time his user-space application creates a temporary file. The idea would be to disable by default the audit files and the auditor must choose which file must be audited and which shouldn't (based on the names of the files).

- **User management through Filesystem interface:** With more development, it should be possible to manage the user entitlement and user management by simply editing a file. Indeed, as LAUXUS can control each IO call interaction, we could create an XML file for the user entitlement. By correctly implementing the back end, we could intercept the information we

want and update accordingly the nodes metadata structure. This process would mean that depending on the user role, the Filesystem interface can be very different (e.g: the authorised user sees the content, the owner user sees the user entitlement and the administrator only see the user database).

- **Audit file visualiser:** Similarly to the above point, we can use the same principle for the auditor. In that way, it becomes easy for him to read the entries. The goal of these last two points is to use the Filesystem to provide the entire interface for pure ease of use.

- **Forensic Analysis:** Lastly, as discussed in the two previous sections, there is a huge opportunity for forensic analysis thanks to the Authenticated Encryption algorithms. The idea would be to alert the administrator when someone wrongly modifies a file or a metadata structure. Furthermore, the different ECC keys used helps forensic to exactly pinpoint which user and on which computer the action has been done.

Chapter 6

Conclusion

This work brought LAUXUS: an auditable and secure Personal Storage that allows lambda users to protect their information against not just the Cloud itself but also malware. We developed and tested a client software that protects personal and private information while still leveraging the advantages of personal cloud technologies. This in itself is already a very interesting application and a good example of the power of SGX Enclaves.

On top of the cited security benefits, LAUXUS embeds innovative technologies to track the user interaction with the Filesystem. The idea is that LAUXUS transform a filesystem into a GDPR compliant one thanks to its purpose aware access that can't be avoided (which means no access can be done without being audited). Furthermore, it provides some strong and fined grained control access policies. As a concrete application example, we used the hospital analogy where the doctors handle so much personal information from the patients. Unfortunately using current technologies and by analysing the current state of the art, it's hard to use a software that proves that these documents are securely stored and are GDPR compliant (only used for legal purposes). This is especially in this situation that LAUXUS comes to the rescue.

We analysed our model against a lot of possible threats at many different levels considering nearly every party as a potentially malicious user. The only trusted point in the model is the Intel SGX Enclave. The model proved to have really strong security and privacy level that even protects against Man-In-The-Computer attacks. Furthermore, our implementation proved to be able to transparently make all of this possible (especially tracking which user has access which files at what time).

On top of that, we implemented our whole model to check its performance. Even though performance is not the primary concern of our work, we wanted to see if it would be a viable solution. We proved that it had a reasonable overhead

compared to using the classical Linux filesystem. By reasonable overhead, we mean that even though it is two times slower than classical Linux filesystem, it doesn't make the filesystem non-usable or feeling slow when used. Although the implementation and the model are not perfect, it is already an advanced work that can, with reasonable work, be transformed into a real, user-friendly software useable by anyone with the correct hardware.

Bibliography

[1] Alexandre Adamski. Overview of intel sgx - part 1, sgx internals, 2018. [Online; accessed 10-Mars-2020].

[2] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'keeffe, D., Stillwell, M. L., et al. {SCONE}: Secure linux containers with intel {SGX}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 689–703.

[3] Baumann, A., Peinado, M., and Hunt, G. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS) 33*, 3 (2015), 1–26.

[4] Burihabwa, D., Felber, P., Mercier, H., and Schiavoni, V. Sgx-fs: Hardening a file system in user-space with intel sgx. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (2018), pp. 67–72.

[5] Chow, R., Golle, P., Jakobsson, M., Shi, E., Staddon, J., Masuoka, R., and Molina, J. Controlling data in the cloud: Outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security* (New York, NY, USA, 2009), CCSW '09, Association for Computing Machinery, p. 85–90.

[6] Dam, M., Guanciale, R., Khakpour, N., Nemati, H., and Schwarz, O. Formal verification of information flow security for a simple arm-based separation kernel. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), pp. 223–234.

[7] Djoko, J. B., Lange, J., and Lee, A. J. Nexus: Practical and secure access control on untrusted storage platforms using client-side sgx. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2019), IEEE, pp. 401–413.

[8] Ekberg, J.-E., Kostiainen, K., and Asokan, N. Trusted execution environments on mobile devices. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), pp. 1497–1498.

[9] European Data Protection Supervisor. Gdpr records of processing activities, 2018. [Online; accessed 20-April-2020].

[10] European Data Protection Supervisor. Gdpr records register, 2018. [Online; accessed 20-April-2020].

[11] Garcia Lopez, P., Montresor, A., Epema, D., Datta, A., Higashino, T., Iamnitchi, A., Barcellos, M., Felber, P., and Riviere, E. Edge-centric computing: Vision and challenges. *SIGCOMM Comput. Commun. Rev. 45*, 5 (Sept. 2015), 37–42.

[12] Goh, E.-J., Shacham, H., Modadugu, N., and Boneh, D. Sirius: Securing remote untrusted storage. In *NDSS* (2003), vol. 3, pp. 131–145.

[13] Guan, L., Liu, P., Xing, X., Ge, X., Zhang, S., Yu, M., and Jaeger, T. Trustshadow: Secure execution of unmodified applications with arm trustzone. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services* (2017), pp. 488–501.

[14] Intel SGX. Sgx api specification, 2018. [Online; accessed 10-May-2020].

[15] Intel SGX. Sgx developer guide, 2018. [Online; accessed 10-Mars-2020].

[16] Jain, P., Desai, S. J., Shih, M.-W., Kim, T., Kim, S. M., Lee, J.-H., Choi, C., Shin, Y., Kang, B. B., and Han, D. Opensgx: An open platform for sgx research. In *NDSS* (2016).

[17] Jang, J. S., Kong, S., Kim, M., Kim, D., and Kang, B. B. Secret: Secure channel between rich execution environment and trusted execution environment. In *NDSS* (2015).

[18] JP Aumasson, Luis Merino. Sgx api specification. [Online; accessed 10-May-2020].

[19] Kevin Poniatowski. Is the stride approach still relevant for threat modeling ?, 2018. [Online; accessed 24-May-2020].

[20] Krahn, R., Trach, B., Vahldiek-Oberwagner, A., Knauth, T., Bhatotia, P., and Fetzer, C. Pesos: Policy enhanced secure object store. In *Proceedings of the Thirteenth EuroSys Conference* (2018), pp. 1–17.

[21] LIBFUSE. Fuse filesystem example. [Online; accessed 10-May-2020].

[22] NGABONZIZA, B., MARTIN, D., BAILEY, A., CHO, H., AND MARTIN, S. Trustzone explained: Architectural features and use cases. In *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)* (2016), IEEE, pp. 445–451.

[23] PONTES, R., BURIHABWA, D., MAIA, F., PAULO, J. A., SCHIAVONI, V., FELBER, P., MERCIER, H., AND OLIVEIRA, R. Safefs: A modular architecture for secure user-space file systems: One fuse to rule them all. In *Proceedings of the 10th ACM International Systems and Storage Conference* (New York, NY, USA, 2017), SYSTOR '17, Association for Computing Machinery.

[24] RAJGARHIA, A., AND GEHANI, A. Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (New York, NY, USA, 2010), SAC '10, Association for Computing Machinery, p. 206–213.

[25] SABT, M., ACHEMLAL, M., AND BOUABDALLAH, A. Trusted execution environment: What it is, and what it is not. *2015 IEEE Trustcom/BigDataSE/ISPA 1* (2015), 57–64.

[26] SELVARAJ SURENTHAR (INTEL). Comparison between arm trustzone and intel sgx enclaves, 2017. [Online; accessed 20-May-2020].

[27] SHAH, A., BANAKAR, V., SHASTRI, S., WASSERMAN, M., AND CHIDAMBARAM, V. Analyzing the impact of GDPR on storage systems. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)* (Renton, WA, July 2019), USENIX Association.

[28] SHASTRI, S., WASSERMAN, M., AND CHIDAMBARAM, V. The seven sins of personal-data processing systems under GDPR. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (Renton, WA, July 2019), USENIX Association.

[29] VANGOOR, B. K. R., TARASOV, V., AND ZADOK, E. To {FUSE} or not to {FUSE}: Performance of user-space file systems. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)* (2017), pp. 59–72.

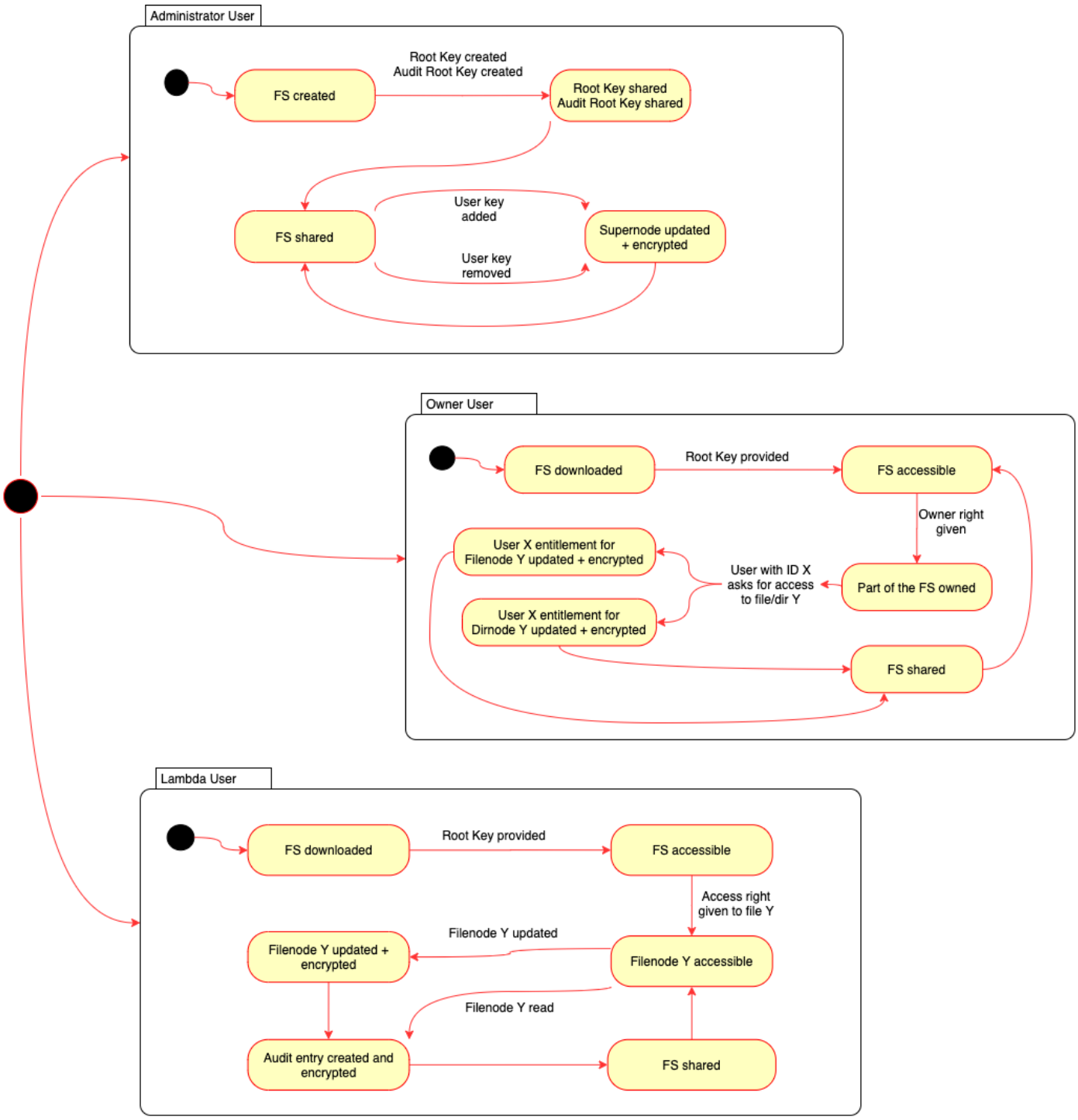
[30] WIKIPEDIA CONTRIBUTORS. Criticism of dropbox — Wikipedia, the free encyclopedia, 2019. [Online; accessed 8-June-2020].

[31] WIKIPEDIA CONTRIBUTORS. Digital rights management — Wikipedia, the free encyclopedia, 2020. [Online; accessed 3-June-2020].

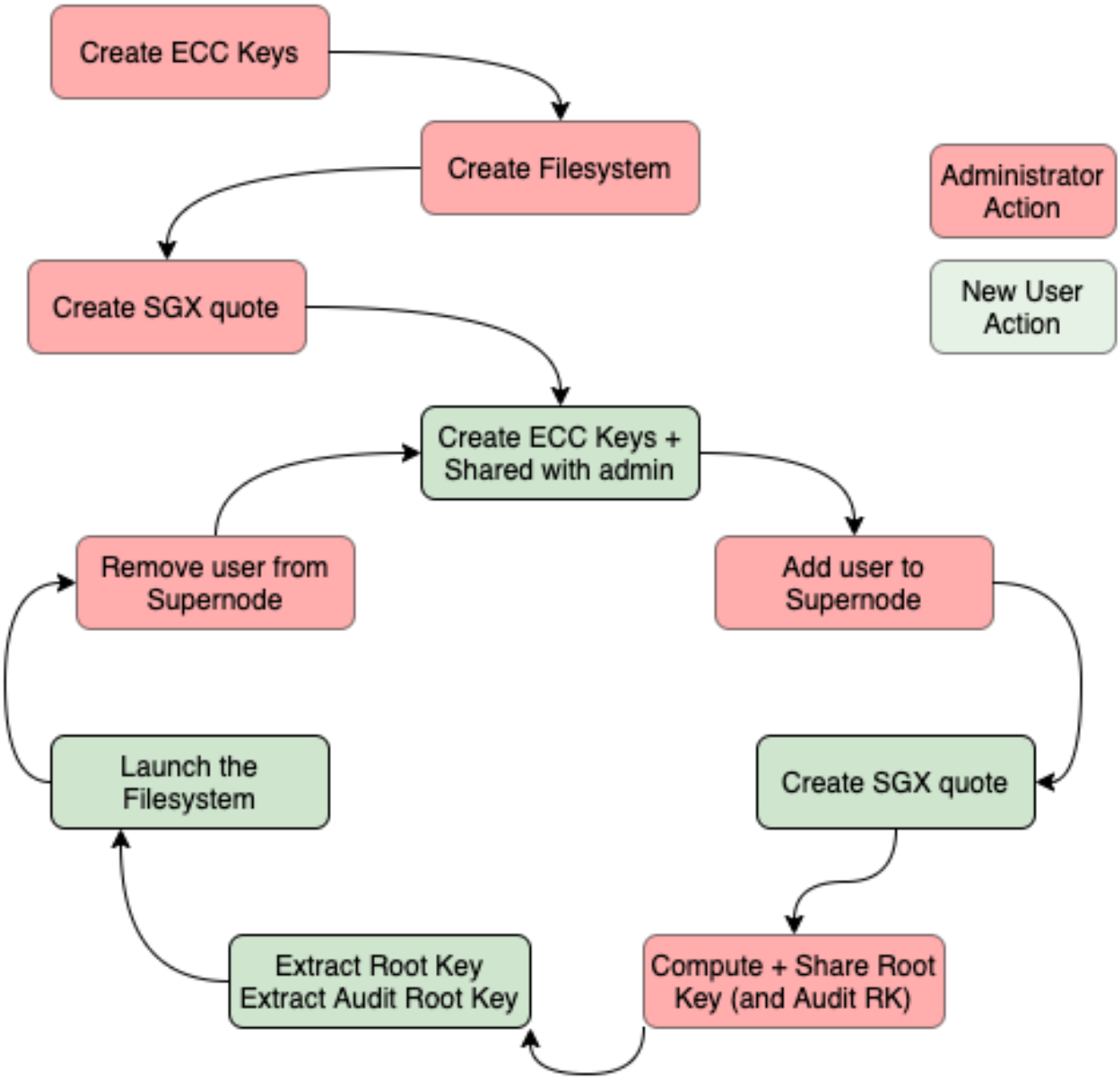
[32] Wikipedia contributors. Stride (security) — Wikipedia, the free encyclopedia, 2020. [Online; accessed 24-May-2020].

Appendices

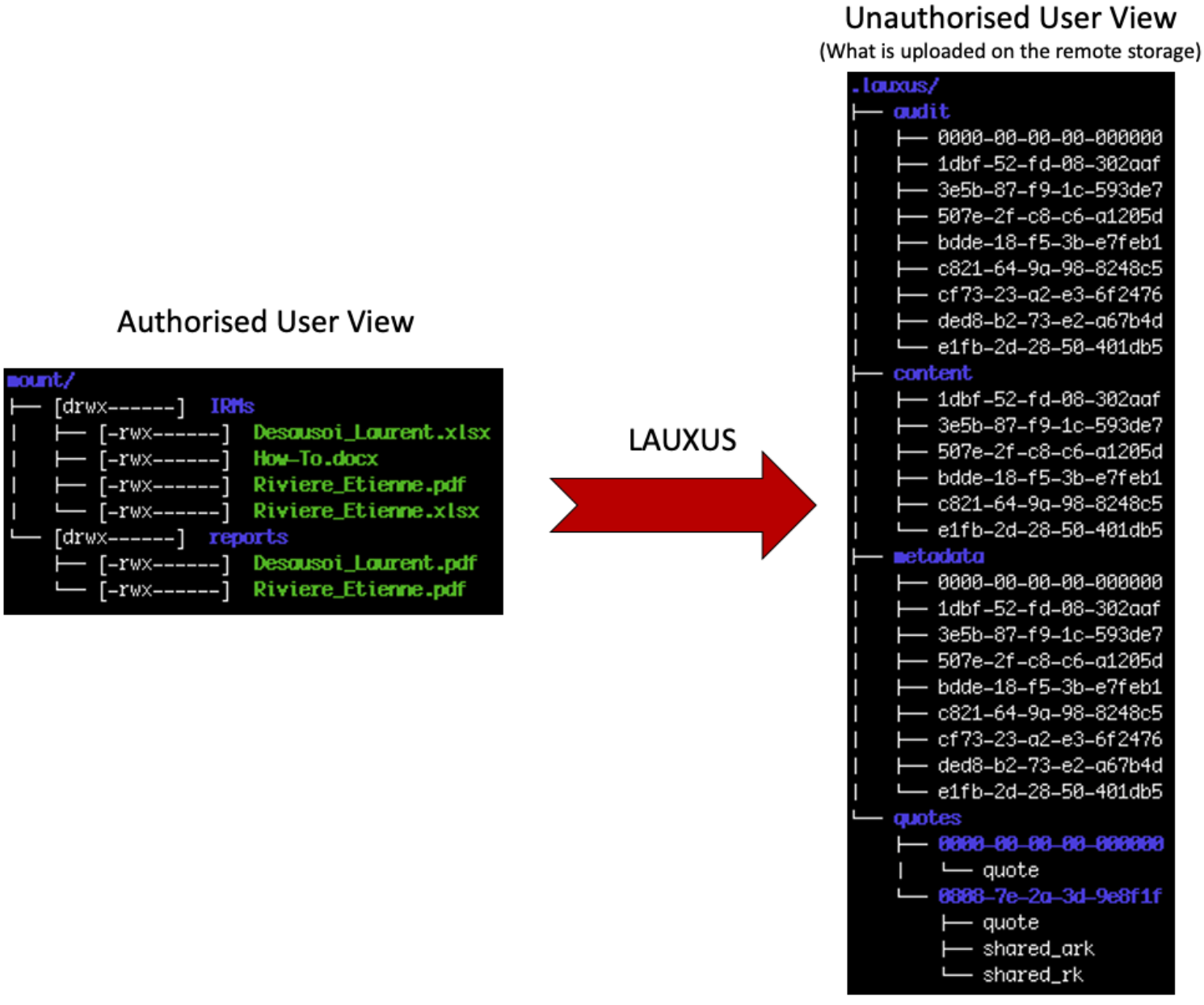
A LAUXUS Life State diagram



B LAUXUS Login Overall process



C LAUXUS Structure in action[1]



[1]Filesystem with two users (administrator and one lambda user).

D Additional Performances

D.1 Writing time VS offset

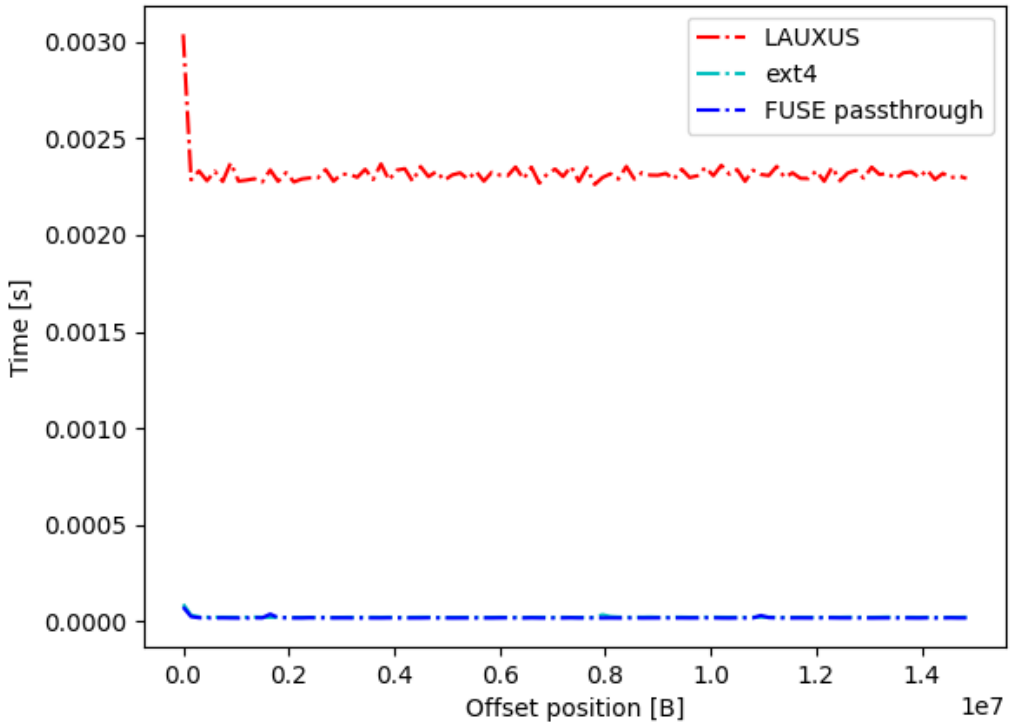


Figure 1: Small offset edition into a 15MB file at different index

From analysing the Section 5.2, we want to be sure that our implementation is correct. In more details, we want to be sure that, for fixed file size, writing a few bytes takes the same amount of time no matter the offset. This expectation seems trivial by analysing our theoretical model but an incorrect implementation could end up giving different results. By analysing the Figure 1, we clearly see that our implementation is correct (for this functionality at least).

E AES-CTR nonce reusability weakness

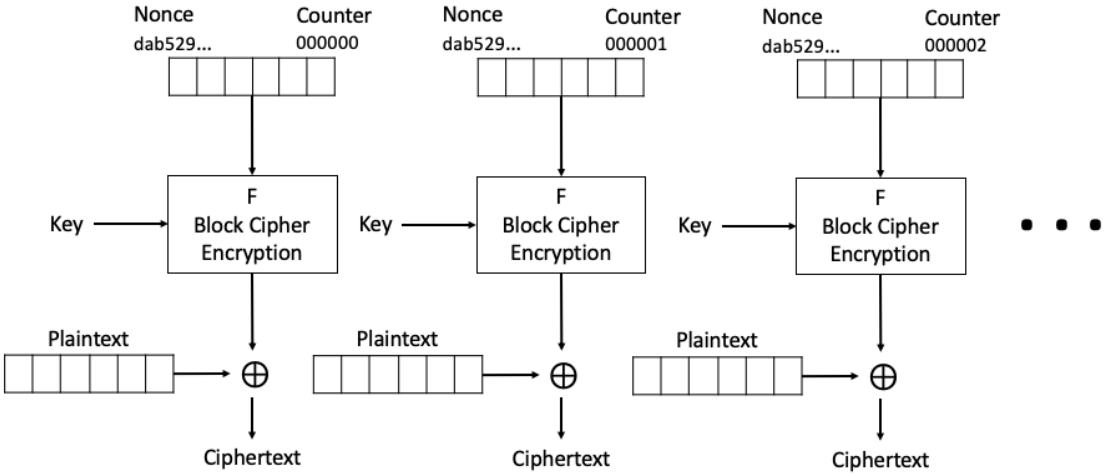


Figure 2: AES CTR encryption

From the above Figure 2, the scheme is mainly build on a complex block cipher F. And we have the following equation:

$$C = P \oplus F(Key, IV) \tag{1}$$

In this situation, if a user reuses the same Key/IV pair, we would obtain the following ciphertexts:

$$\begin{aligned} C_1 &= P_1 \oplus F(Key, IV) \\ C_2 &= P_2 \oplus F(Key, IV) \end{aligned} \tag{2}$$

This implies that an attacker can then compute with ease:

$$C_1 \oplus C_2 = P_1 \oplus P_2 \tag{3}$$

And derive the values of the two plaintexts xored together without even knowing neither the key or the IV.

F Source code insights

LAUXUS[2] is written fully in C++ but using only using few C++ functionalities (e.g: we don't use template, etc). In the end, it has over six thousands lines of code

[2]Github repository: https://github.com/MiniLau/Lauxus

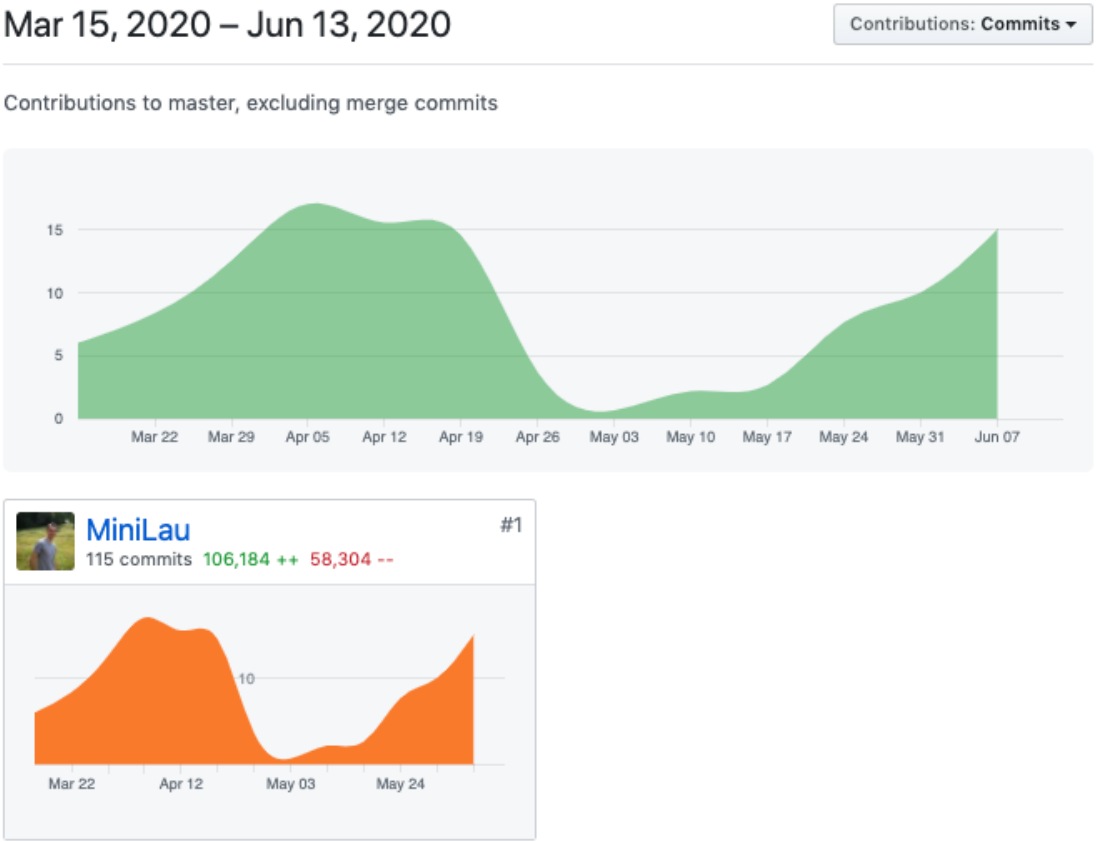


Figure 3: Github code evolution

with unit testing included. We designed our code to be split into two main folders. One which holds all the source code of the trusted application (code run inside the Enclave) and one for the source code of the untrusted application (code run outside the Enclave). I found that this structure was best suited as at the end, there were way too many files to have them all inside a single folder. Plus, mixing both codes in the same folder was extremely confusing as we no longer understand where the code we write will be running (either in the Enclave or the untrusted application).

By observing Figure 3, we see that the base code of the application required two highly intensive months. Furthermore, I wasn't able to track before March because my git was corrupted and I had to delete it all and solve the issues by hand. On top of that, these two months doesn't take into account the time taken to: understand the required design, understand how to develop with *FUSE* and SGX Enclaves and understanding all the other smaller requirements.

However, I like to point that before working on LAUXUS, I worked on some smaller Enclave project to understand how Enclaves worked. I found that SGX

development doesn't have at all an easy learning curve even though I have a lot of experience in developing. I realise that because even after a few months working on LAUXUS, there were still things that I didn't understand about Enclaves (mainly on how to write clean code and properly using ECALLS and OCALLS).

G Source code Unit Testing

To diagnose, debug and prove that our software was working well, we used, as anticipated, the unit testing. In more details, we used the C++ Catch2 framework.

The unit testing was mandatory in our work to prove that there were no memory leaks. Indeed, as we discussed in the report, SGX Enclaves are using a limited amount of memory. Furthermore, as the filesystem might be running for a long time, even a very small memory leak can become disastrous. Not only small leaks but also bigger ones, as we are handling huge amount of data, not freeing them would make our software unusable.

Upon a lot of research, I wasn't able to find a framework to do some unit testing directly inside the Enclave. Thus, I simulated the minimum required calls to test my code. Of course, this means that when we are using the unit testing we don't obtain the same results as when we are using the Enclave (e.g: simulated encryption is rudimental). This doesn't matter because this simulation layer is only important to prove that our code works and that there are no memory leaks. As long as our simulation layer works like the Enclave (in term of return results and memory usage), the unit tests are relevant.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**